REPORT:

ML PP4 Report

DATASET: artsmall

KERNEL: LINEAR

Value of **alpha** after convergence: **143.83708956072442**

Value of **beta** after convergence: **4.138171016965101**

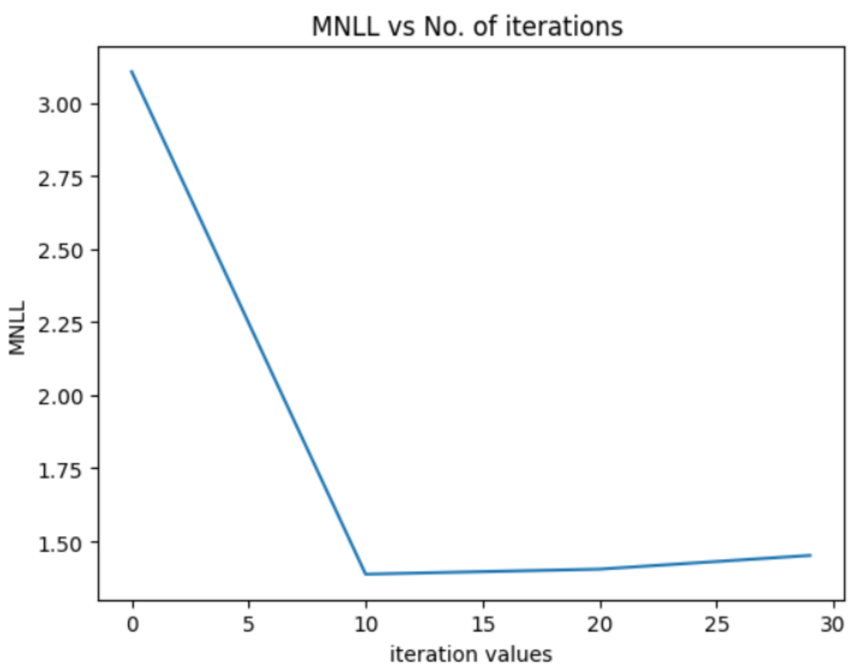
no of iterations: **30**

MSE: **0.7096827392160092**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
convergence condition met.
LINEAR KERNEL: Final value of alpha: 143.83708956072442, beta: 4.138171016965101 and no of iterations: 30
MSE: 0.7096827392160092

Process finished with exit code 0
```

MNLL vs Iterations Plot:



KERNEL: RBF

Value of **alpha** after convergence: **0.42420765**

Value of **beta** after convergence: **6.4933422**

Value of **s** after convergence: **16.21435596**

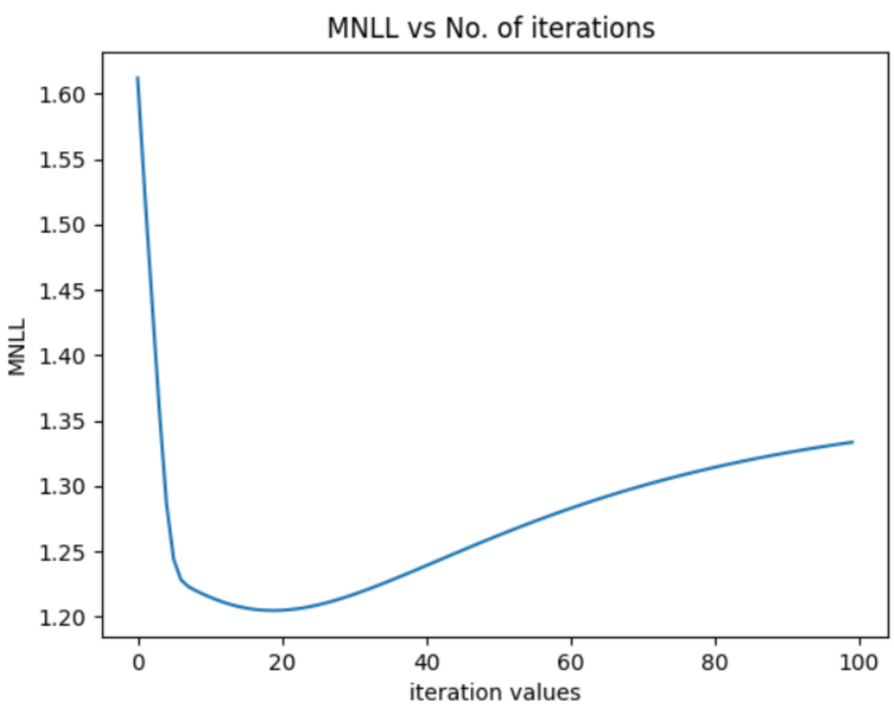
no of iterations: **100**

MSE: **0.6780432335752934**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
convergence condition met.
RBF KERNEL: Final value of alpha: [[0.42420765]], beta: [[6.4933422]], s: [[16.21435596]] and no of iterations: 100
MSE: 0.6780432335752934

Process finished with exit code 0
```

MNLL vs Iterations Plot:



<u>DATASET: crime</u>

<u>KERNEL: LINEAR</u>

Value of **alpha** after convergence: **353.3792778969971**

Value of **beta** after convergence: **2.604925838601953**

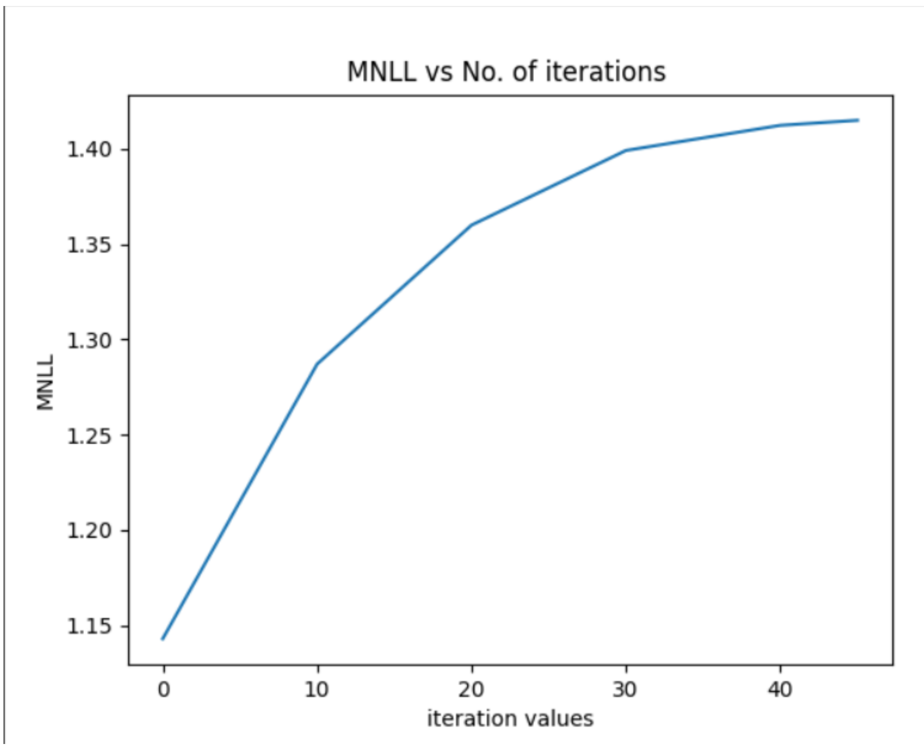
no of iterations: **46**

MSE**: 0.5026610661299366**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
convergence condition met.
LINEAR KERNEL: Final value of alpha: 353.3792778969971, beta: 2.604925838601953 and no of iterations: 46
MSE: 0.5026610661299366

Process finished with exit code 0
```

<u>MNLL vs Iterations Plot:</u>



KERNEL: RBF

Value of **alpha** after convergence: **0.65813614**

Value of **beta** after convergence**: 2.76193131**

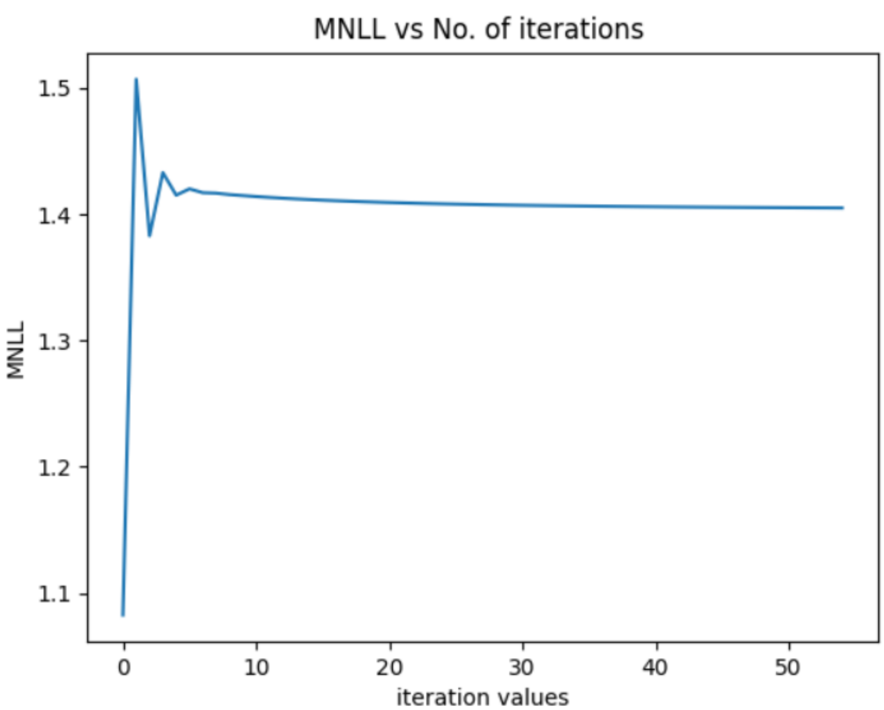
Value of **s** after convergence: **21.25812348**

no of iterations: **55**

MSE: **0.49932365931240286**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
convergence condition met.
RBF KERNEL: Final value of alpha: [[0.65813614]], beta: [[2.76193131]], s: [[21.25812348]] and no of iterations: 55
MSE: 0.49932365931240286
```

MNLL vs Iterations Plot:



DATASET: housing

KERNEL: LINEAR

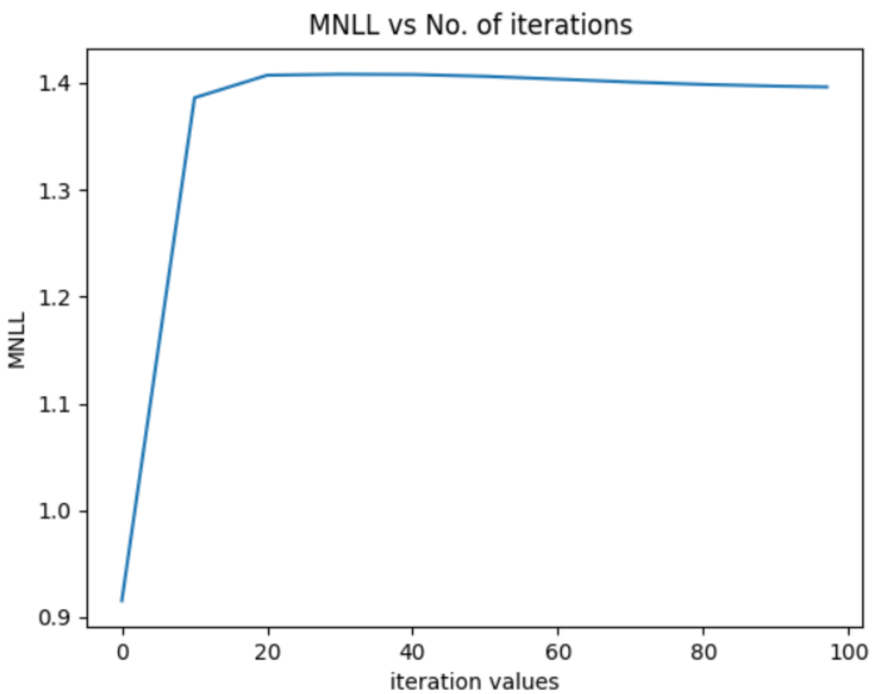
Value of **alpha** after convergence: **21.243840386172003**

Value of **beta** after convergence: **3.9957954472355377**

no of iterations: **98**

MSE: **0.2883918769478013**

MNLL vs Iterations Plot:



KERNEL: RBF

Value of **alpha** after convergence: **0.31272331**

Value of **beta** after convergence: **12.7608653**

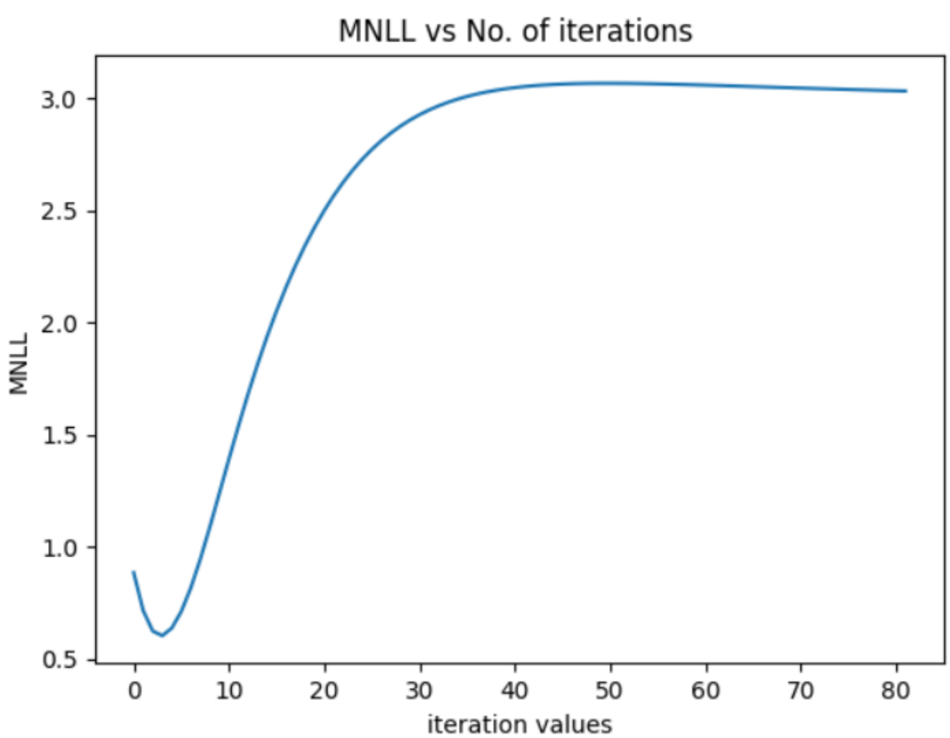
Value of **s** after convergence: **4.80903028**

no of iterations: **82**

MSE: **0.17759333674640404**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
convergence condition met.
RBF KERNEL: Final value of alpha: [[0.31272331]], beta: [[12.7608653]], s: [[4.80903028]] and no of iterations: 82
MSE: 0.17759333674640404
```

<u>MNLL vs Iterations Plot:</u>



DATASET: 1D

KERNEL: LINEAR

Value of **alpha** after convergence: **2.3836990764543935**

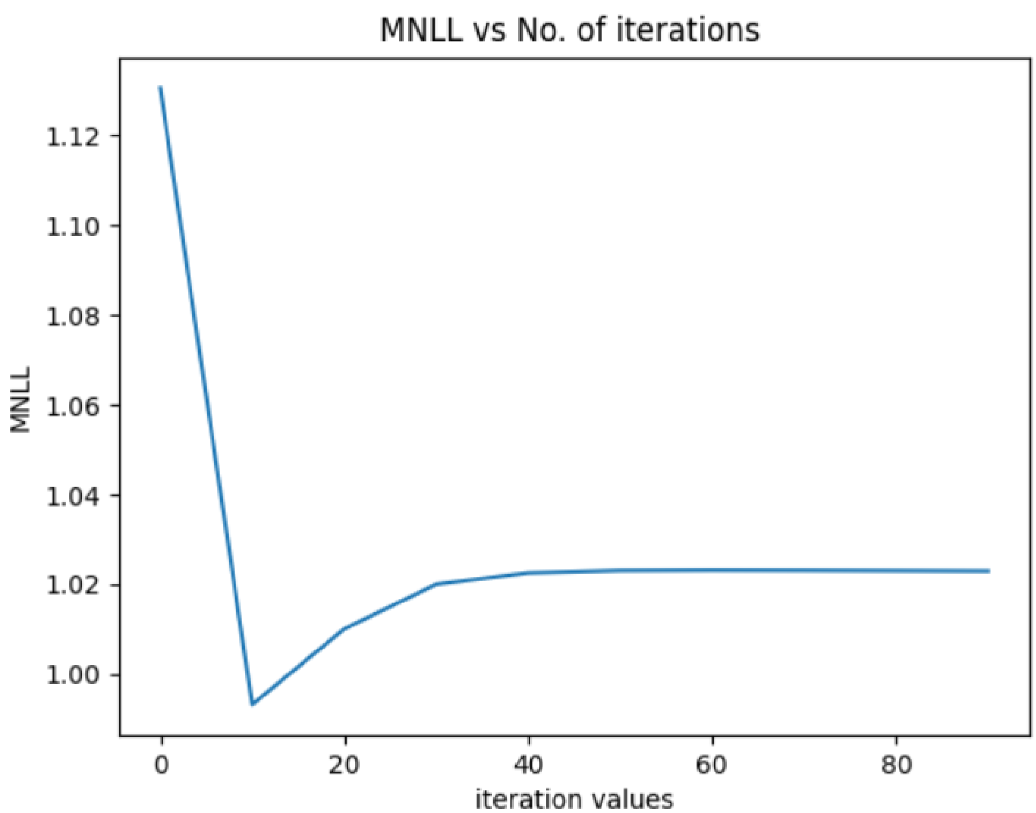
Value of **beta** after convergence: **1.926161713110879**

no of iterations: **100**

MSE: **0.41146991047994314**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
LINEAR KERNEL: Final value of alpha: 2.3836990764543935, beta: 1.926161713110879 and no of iterations: 100
MSE: 0.41146991047994314
```

MNLL vs Iterations Plot:



KERNEL: RBF

Value of **alpha** after convergence: **0.95347473**

Value of **beta** after convergence: **1.85243098**

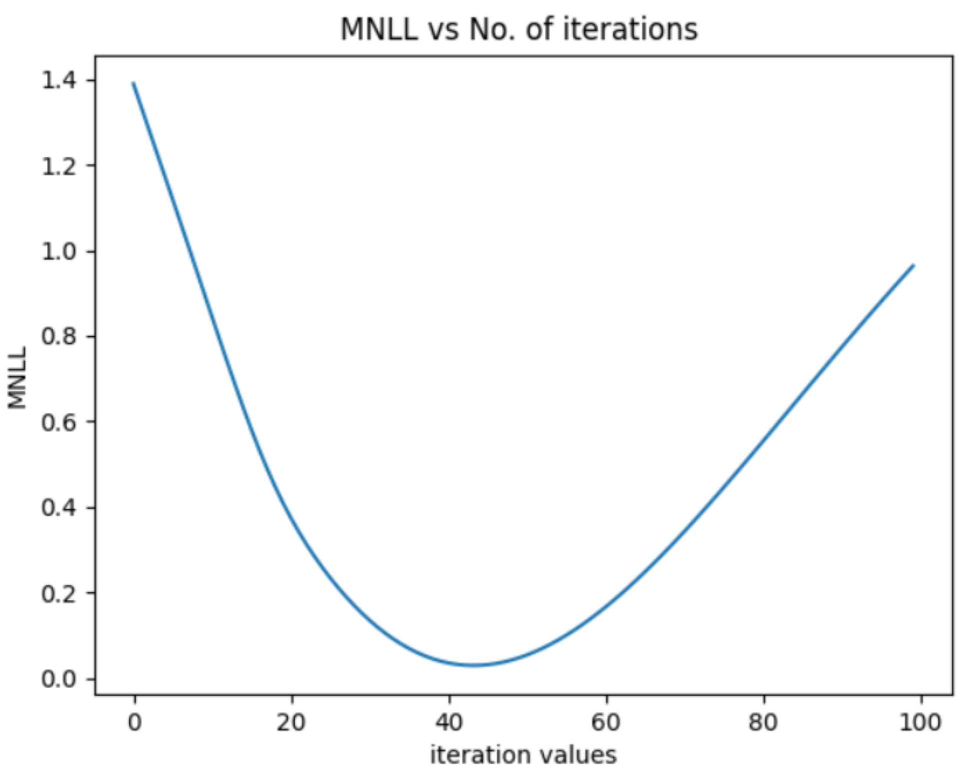
Value of **s** after convergence: **4.26539792**

no of iterations: **35**

MSE: **0.42856226814020515**

```
"C:\Users\Abhishek Sharma\PycharmProjects\pp4\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/pp4/main.py"
convergence condition met.
RBF KERNEL: Final value of alpha: [[0.95347473]], beta: [[1.85243098]], s: [[4.26539792]] and no of iterations: 35
MSE: 0.42856226814020515
```

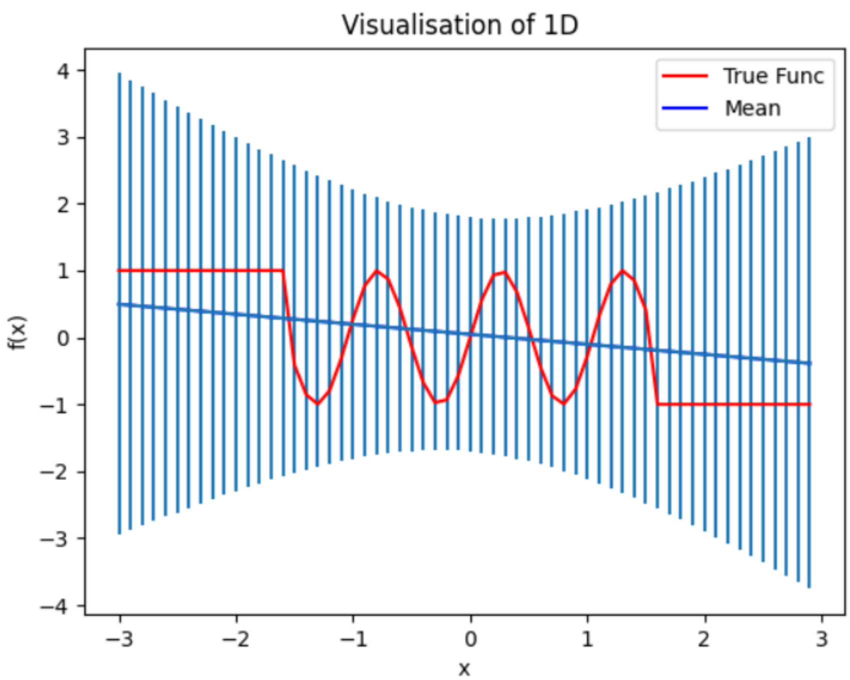
<u>MNLL vs Iterations Plot:</u>



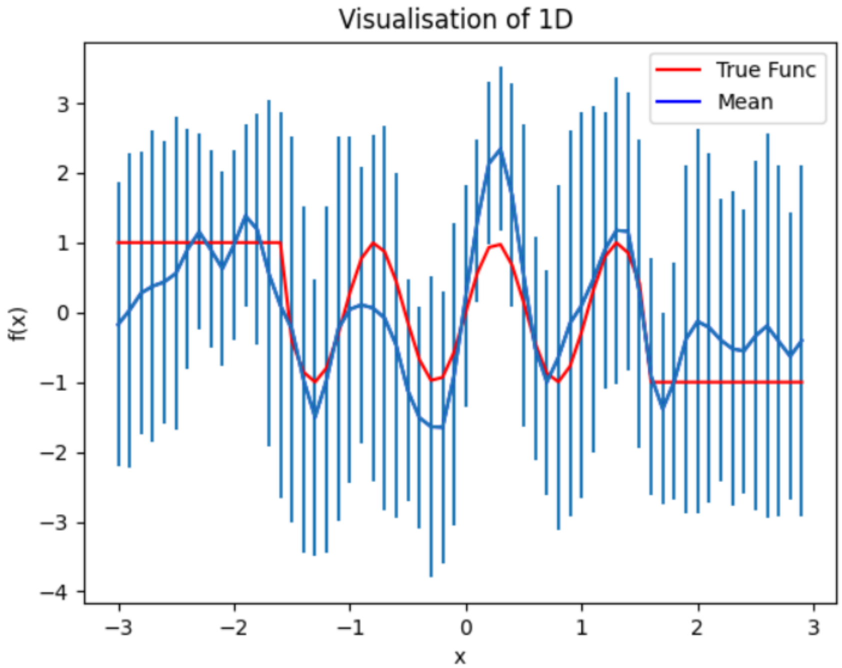
VISUALISATION OF PERFORMANCE OF 1D DATASET

-PLOT OF TRUE FUNCTION VS THE MEAN

LINEAR KERNEL:



RBF:



Comparison to Bayesian Linear Regression

LINEAR KERNEL:

DATASET	**MSE**	**ALPHA**	**BETA**
Crime	0.5026610661299366	353.3792778969971	2.604925838601953
Housing	0.2883918769478013	21.243840386172003	3.9957954472355377
Artsmall	0.7096827392160092	143.83708956072442	4.138171016965101
1D	0.41146991047994314	2.3836990764543935	1.926161713110879

RBF KERNEL:

DATASET	**MSE**	**ALPHA**	**BETA**
Crime	0.49932365931240286	0.65813614	2.76193131
Housing	0.17759333674640404	0.31272331	12.7608653
Artsmall	0.7096827392160092	0.42420765	6.4933422
1D	0.42856226814020515	0.95347473	1.85243098

The values of alpha and beta in case of Linear Kernel are quite close to the corresponding values in the table given in the assignment.
In case of RBF kernel, the values are different. This is because Linear and RBF are both different feature spaces and there need not be any relation between the two.

The test set MSE in both Linear and RBF is almost similar to the values given in the table. This shows that the MSE is not affected by the choice of Kernel.

Discussion of results

<u>Q1.</u> Are the BLR and GP with linear kernel behaving similarly w.r.t. α,β, MSE as expected?

Ans. As seen from the tables above, the values of alpha, beta and MSE are almost similar to the expected values. Thus, the BLR and GP with linear kernel behave similarly w.r.t. α,β, MSE.

Q2. How does the performance of GP compare when changing RBF vs. linear kernel?

Ans. The performances with RBF and Linear kernel vary wrt the running times and prediction accuracy.

The RBF kernel takes much longer to run than a Linear kernel. Also, the no of iterations for convergence of Log(Ev) in Linear are less compared to that of RBF.

However, in terms of prediction accuracy, as observed in the visualisation of performance of 1D dataset, the prediction mean is more close to the true function in case of RBF kernel and not much close in case of a linear kernel. This shows that RBF provides more accurate predictions.

Q3. What are potential advantages or disadvantages of each method?

Ans. The advantages of a linear kernel is that it provides results faster and therefore can be beneficial in case of large datasets.

A disadvantage of Linear is that it might not provide the most accurate predictions, though, a good prediction nonetheless.

Advantage of RBF is that it provides more accurate solutions and is therefore useful in models that require high precision and accuracy.

The disadvantage is the longer running times which might not be beneficial with large datasets.

<u>PP4 README</u>

Info about the code:

1. **The code is divided into three sections: Part 1: GP for Linear Kernel, Part 2: GP for RBF Kernel and Part 3: Visualisation of performance of 1D dataset.**
2. **Out of these 3 sections of code, Part 1 is left uncommented and the remaining three are commented.**

Steps to run the code:

1. **Change the paths of the datasets according to the paths on SICE servers.**

```
train_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-1D.csv"
trainR_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-1D.csv"
train_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-artsmall.csv"
trainR_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-artsmall.csv"
train_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-crime.csv"
trainR_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-crime.csv"
train_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-housing.csv"
trainR_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-housing.csv"
test_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-1D.csv"
testR_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-1D.csv"
test_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-artsmall.csv"
testR_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-artsmall.csv"
test_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-crime.csv"
testR_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-crime.csv"
test_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-housing.csv"
testR_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-housing.csv"
```

2. **The default dataset chosen is artsmall. In order to test the code on other datasets, just replace the dataset name in all 4 file paths from the list of names given.**

```
# read train data
XTrainData = np.array(gp.readData(train_artsmall))
XTrainData = XTrainData.astype(np.float64)

# train labels t
XTrainData_label = np.array(gp.readData(trainR_artsmall))
XTrainData_label = XTrainData_label.astype(np.float64)

# read test data
YTestData = np.array(gp.readData(test_artsmall))
YTestData = YTestData.astype(np.float64)

# test labels ti
YTestData_label = np.array(gp.readData(testR_artsmall))
YTestData_label = YTestData_label.astype(np.float64)
```

3. **Part 1: GP with Linear Kernel is uncommented, and can be run right after step 1.**

```
#==========================================GP WITH LINEAR KERNEL==================================================
# obtain final values of alpha, beta and iterations after gradient ascent
alpha, beta, iterations, alphaList, betaList,iterationValues = gp.modelSelection_Linear(K_Linear, XTrainData_label, alpha, beta, eta, XTrainData)
print('LINEAR KERNEL: Final value of alpha: ' + str(alpha) + ', beta: ' + str(beta) + ' and no of iterations: ' + str(
    iterations + 1))

# EVALUATION

# calculating MNLL at every 10th iteration
MSE,MNLL_List = gp.evaluation_Linear(alphaList,betaList,iterationValues,K_Linear,XTrainData,YTestData,XTrainData_label,YTestData_label,'Linear')

print('MSE:',MSE)

plt.plot(iterationValues,MNLL_List)
plt.xlabel("iteration values")
plt.ylabel("MNLL")
plt.title("MNLL vs No. of iterations ")
plt.show()
```

4. **In order to run Part 2: GP with RBF Kernel, just uncomment Part 2 section and comment the Part 1 section and execute the code.**

```
# # # ========================================GP WITH RBF KERNEL======================================================
# # obtain final values of alpha, beta, s and iterations after gradient ascent
# alpha, beta, s, iterations,alphaList,betaList,sList,iterationValues = gp.modelSelection_RBF(K_RBF,XTrainData_label,alpha,beta,s,eta,XTrainData)
#
# print('RBF KERNEL: Final value of alpha: ' + str(alpha) + ', beta: ' + str(beta) + ', s: ' + str(
#     s) + ' and no of iterations: ' + str(iterations + 1))
# #
# ##EVALUATION
#
# # calculating MNLL at every 10th iteration
# MSE, MNLL_List = gp.evaluation_RBF(alphaList,betaList,sList,iterationValues,XTrainData,YTestData,XTrainData_label, YTestData_label,'RBF')
#
# print('MSE:', MSE)
#
# plt.plot(iterationValues,MNLL_List)
# plt.xlabel("iteration values")
# plt.ylabel("MNLL")
# plt.title("MNLL vs No. of iterations ")
# plt.show()
```

5. **To run Part 3: Visualisation of performance of 1D dataset, just uncomment the section and comment the above two sections and execute the code.**

```
# ==========================================VISUALISATION OF 1D PERFORMANCE===============================
# alpha_1D = 1
# beta_1D = 1
# s_1D = 0.1
#
# # Linear
# gp.visualise1D(alpha_1D,beta_1D,s_1D,'Linear')
#
# # RBF
# gp.visualise1D(alpha_1D,beta_1D,s_1D,'RBF')
```

6. **The plots will be automatically shown at the end of the execution.**

CODE:

```
# import necessary libraries
import csv
import time

# import time
import numpy as np

# import statistics
#
import math

# import operator
import matplotlib.pyplot as plt

train_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-1D.csv"
trainR_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-
1D.csv"
train_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-
artsmall.csv"
trainR_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-
artsmall.csv"
train_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-
crime.csv"
trainR_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-
crime.csv"
train_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/train-
housing.csv"
trainR_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/trainR-
housing.csv"
test_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-1D.csv"
testR_1D = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-1D.csv"
test_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-
artsmall.csv"
testR_artsmall = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-
artsmall.csv"
test_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-
crime.csv"
testR_crime = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-
crime.csv"
test_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/test-
housing.csv"
testR_housing = "C:/Users/Abhishek Sharma/Downloads/pp4data/pp4data/testR-
housing.csv"


class GP:
    # read data
    def readData(self, path):
        file = open(path)
        csvReader = csv.reader(file)
        data = []
        for row in csvReader:
            data.append(row)
        file.close()
        return data
```

```python
def createKMatrix(self, data):
    ## creating Linear Kernel Matrix K(X,X)
    K = np.empty((len(data), len(data)))
    for i in range(len(data)):
        x1 = data[i]
        for j in range(len(data)):
            x2 = data[j]
            K[i][j] = np.dot(x1.T, x2) + 1
    return K

def createKMatrixFor_vT(self, trainData, testData):
    K = np.empty((len(testData), len(trainData)))
    for i in range(len(testData)):
        x1 = testData[i]
        for j in range(len(trainData)):
            x2 = trainData[j]
            K[i][j] = np.dot(x1.T, x2) + 1
    return K

def createK_RBFMatrixFor_vT(self, trainData, testData,s):
    K = np.empty((len(testData), len(trainData)))
    for i in range(len(testData)):
        x1 = testData[i]
        for j in range(len(trainData)):
            x2 = trainData[j]
            exp_term = math.pow(np.linalg.norm(x1 - x2), 2)
            value = np.exp(-0.5 * (exp_term) / (math.pow(s, 2)))
            K[i][j] = value
    return K

def createRBF_KMatrix(self, data, s):
    # creating RBF Kernel Matrix K(X,X)
    K_RBF = np.empty((len(data), len(data)))
    for i in range(len(data)):
        x1 = data[i]
        for j in range(len(data)):
            x2 = data[j]
            exp_term = math.pow(np.linalg.norm(x1 - x2), 2)
            value = np.exp(-0.5 * (exp_term) / (math.pow(s, 2)))
            K_RBF[i][j] = value
    return K_RBF

def calculateCnMatrix(self, K, alpha, beta):
    # now create C(X,X) = 1/Beta * I + 1/alpha * K(X,X) FOR LINEAR KERNEL
    # Identity matrix for C
    I = np.eye(len(K))
    Cn = ((1 / beta) * I) + ((1 / alpha) * K)
    return Cn

def calc_dCn_Alpha(self, alpha, K_Linear):
    # der of Cn wrt alpha = -1/alpha**2 * K
    dCN_dAlpha = -(1 / alpha) ** 2 * K_Linear

    return dCN_dAlpha

def calc_dCn_Beta(self, beta, I_Linear):
```

```
        # der of Cn wrt beta = -1/beta**2 * I
        dCN_dBeta = -(1 / beta) ** 2 * I_Linear

        return dCN_dBeta

    def calc_dCn_s(self, data, s):
        dCn_ds = np.empty((len(data), len(data)))
        for i in range(len(data)):
            x1 = data[i]
            for j in range(len(data)):
                x2 = data[j]
                dCn_ds[i][j] = np.exp(-0.5 * np.linalg.norm(x1 - x2)**2 / (s
** 2)) * (np.linalg.norm(x1 - x2)**2) * 1/ (s ** 3)
        return dCn_ds

    def calc_der_LogEv(self, Cn, dCn, t):
        dLogEv = -0.5 * np.trace((np.linalg.inv(Cn)) @ dCn) + 0.5 * t.T @
(np.linalg.inv(Cn)) @ dCn @ (
            np.linalg.inv(Cn)) @ t
        return dLogEv

    def modelSelection_Linear(self, K, t, alpha, beta, eta, data):
        I = np.eye(len(K))

        # Cn matrix for linear
        Cn = self.calculateCnMatrix(K, alpha, beta)
        N = len(data)
        LogEv_Old = -(N / 2) * np.log(2 * math.pi) - 0.5 *
np.log(np.linalg.det(Cn)) - 0.5 * t.T @ (
            np.linalg.inv(Cn)) @ t

        dCnAlpha = self.calc_dCn_Alpha(alpha, K)
        dCnBeta = self.calc_dCn_Beta(beta, I)
        dLogEv_alpha = self.calc_der_LogEv(Cn, dCnAlpha, t)
        dLogEv_beta = self.calc_der_LogEv(Cn, dCnBeta, t)

        alphaList = []
        betaList = []
        iterationValues = []

        for i in range(0, 100):
            # for alpha
            a = np.log(alpha)
            a = a + eta * dLogEv_alpha * alpha
            alpha = np.exp(a)

            # for beta
            b = np.log(beta)
            b = b + eta * dLogEv_beta * beta
            beta = np.exp(b)

            Cn = self.calculateCnMatrix(K, alpha, beta)

            dCnAlpha = self.calc_dCn_Alpha(alpha, K)
            dCnBeta = self.calc_dCn_Beta(beta, I)
            dLogEv_alpha = self.calc_der_LogEv(Cn, dCnAlpha, t)
            dLogEv_beta = self.calc_der_LogEv(Cn, dCnBeta, t)
```

```
            LogEv_New = -(N / 2) * np.log(2 * math.pi) - 0.5 *
np.log(np.linalg.det(Cn)) - 0.5 * t.T @ (
                np.linalg.inv(Cn)) @ t
            # to select values of alpha,beta at every 10th iteration
            if i % 10 == 0:
                alphaList.append(float(alpha))
                betaList.append(float(beta))
                iterationValues.append(i)

            # print('i: ' + str(i))
            # print('(LogEv_New-LogEv_Old)/abs(LogEv_Old): ' + str((LogEv_New
- LogEv_Old) / abs(LogEv_Old)))

            if (LogEv_New - LogEv_Old) / abs(LogEv_Old) <= 0.00001:
                print('convergence condition met.')
                # to add values at last iteration
                alphaList.append(float(alpha))
                betaList.append(float(beta))
                iterationValues.append(i)
                return float(alpha), float(beta), i, alphaList, betaList,
iterationValues

            LogEv_Old = LogEv_New

        return float(alpha), float(beta), i, alphaList, betaList,
iterationValues

    def modelSelection_RBF(self, K_RBF, t, alpha, beta, s, eta, data):
        I = np.eye(len(K_RBF))
        N = len(data)

        # Cn matrix for RBF
        Cn_RBF = self.calculateCnMatrix(K_RBF, alpha, beta)

        # calculate LogEv using Cn_RBF
        LogEv_Old = -(N / 2) * np.log(2 * math.pi) - 0.5 *
np.log(np.linalg.det(Cn_RBF)) - 0.5 * t.T @ (
            np.linalg.inv(Cn_RBF)) @ t

        # calculating der of Cn wrt hyper parameters
        dCnAlpha = self.calc_dCn_Alpha(alpha, K_RBF)
        dCnBeta = self.calc_dCn_Beta(beta, I)
        dCnS = self.calc_dCn_s(data, s)

        # calculating der of Log(Ev) wrt hyper parameters
        dLogEv_alpha = self.calc_der_LogEv(Cn_RBF, dCnAlpha, t)
        dLogEv_beta = self.calc_der_LogEv(Cn_RBF, dCnBeta, t)
        dLogEv_s = self.calc_der_LogEv(Cn_RBF, dCnS, t)

        alphaList = []
        betaList = []
        sList = []
        iterationValues = []

        for i in range(0, 100):
            # for alpha
```

```
            a = np.log(alpha)
            a = a + eta * dLogEv_alpha * alpha
            alpha = np.exp(a)

            # for beta
            b = np.log(beta)
            b = b + eta * dLogEv_beta * beta
            beta = np.exp(b)

            # for s
            c = np.log(s)
            c = c + eta * dLogEv_s * s
            s = np.exp(c)

            # re-calculate RBF Kernel with updated s value
            K_RBF = self.createRBF_KMatrix(data, s)

            Cn_RBF = self.calculateCnMatrix(K_RBF, alpha, beta)

            # calculating derv of Cn wrt updated hyper parameters
            dCnAlpha = self.calc_dCn_Alpha(alpha, K_RBF)
            dCnBeta = self.calc_dCn_Beta(beta, I)
            dCnS = self.calc_dCn_s(data, s)

            # calculating derv of Log(Ev) wrt updated hyper parameters
            dLogEv_alpha = self.calc_der_LogEv(Cn_RBF, dCnAlpha, t)
            dLogEv_beta = self.calc_der_LogEv(Cn_RBF, dCnBeta, t)
            dLogEv_s = self.calc_der_LogEv(Cn_RBF, dCnS, t)

            LogEv_New = -(N / 2) * np.log(2 * math.pi) - 0.5 *
np.log(np.linalg.det(Cn_RBF)) - 0.5 * t.T @ (
                np.linalg.inv(Cn_RBF)) @ t

            # to select values of alpha,beta at every 10th iteration
            if i % 10 == 0:
                alphaList.append(float(alpha))
                betaList.append(float(beta))
                sList.append(float(s))
                iterationValues.append(i)

            # print('i: ' + str(i))
            # print('(LogEv_New-LogEv_Old)/abs(LogEv_Old): ' + str((LogEv_New
- LogEv_Old) / abs(LogEv_Old)))

            if (LogEv_New - LogEv_Old) / abs(LogEv_Old) <= 0.00001:
                print('convergence condition met.')
                alphaList.append(float(alpha))
                betaList.append(float(beta))
                sList.append(float(s))
                iterationValues.append(i)
                return alpha, beta, s, i, alphaList, betaList, sList,
iterationValues
            # print('LogEv_Old:',LogEv_Old)
            # print('LogEv_New',LogEv_New)
            LogEv_Old = LogEv_New

            alphaList.append(float(alpha))
```

```python
            betaList.append(float(beta))
            sList.append(float(s))
            iterationValues.append(i)

        return alpha, beta, s, i, alphaList, betaList, sList, iterationValues

    def calculate_mi_vi(self, XTrainData, YTestData, XTrainData_label, Cn,
alpha, beta,s,kernalType):

        if kernalType =='Linear':
            # creating c, vT
            # c=C(xN+1,xN+1)
            # first calculate K matrix, and then calculate c
            K_c = self.createKMatrix(YTestData)
            c = np.array(self.calculateCnMatrix(K_c, alpha, beta))
            # c = np.diag(c)
            # print('c shape: ', c.shape)

            # vT vector
            # vT=(C(x1,xN+1),...,C(xN,xN+1))
            K_vT = self.createKMatrixFor_vT(XTrainData, YTestData)
            C_xN_xNPlus1 = (1 / alpha) * K_vT
            vT = C_xN_xNPlus1
            # print('vT shape: ', vT.shape)
            # print('np.linalg.inv(Cn) shape: ' +
str(np.linalg.inv(Cn).shape))
            # print('XTrainData_label shape: ',XTrainData_label.shape)
        else:
            K_c = self.createRBF_KMatrix(YTestData,s)
            c = np.array(self.calculateCnMatrix(K_c, alpha, beta))
            # c = np.diag(c)
            # print('c shape: ', c.shape)
            K_vT = self.createK_RBFMatrixFor_vT(XTrainData,YTestData,s)
            C_xN_xNPlus1 = (1 / alpha) * K_vT
            vT = C_xN_xNPlus1


        # calculate mi and vi
        mi = np.array(vT @ np.linalg.inv(Cn) @ XTrainData_label)
        vi = c - vT @ np.linalg.inv(Cn) @ vT.T
        vi = np.diag(vi)
        # print(vi)
        # print('mi shape: ' + str(mi.shape))
        # print('vi shape: ' + str(vi.shape))

        return mi, vi

    def calculateMSE(self, mi, ti):
        squareError_Sum = 0.0

        for i in range(len(mi)):
            squareError_i = float((mi[i] - ti[i]) ** 2)
            squareError_Sum += squareError_i

        MSE = squareError_Sum / len(mi)
        return MSE
```

```
    def calculateMNLL(self, mi, vi, ti):
        NLL = []
        for i in range(len(mi)):
            NLL_i = ((1 / (vi[i] * math.sqrt(2 * math.pi))) * np.exp((-1 / 2)
* (((ti[i] - mi[i]) / vi[i]) ** 2)))
            NLL.append(float(-np.log(NLL_i)))
        NLL = np.array(NLL)
        # print('NLL shape', NLL.shape)
        return NLL

    def evaluation_Linear(self, alphaList, betaList, iterationValues,
K_Linear, XTrainData, YTestData, XTrainData_label,
                          YTestData_label,kernelType):
        MNLL_List = []
        for i in range(len(alphaList)):
            alpha = alphaList[i]
            beta = betaList[i]

            # calculate Cn matrix for Linear kernel to calculate mu and sigma
            Cn = self.calculateCnMatrix(K_Linear, alpha, beta)

            # calculating mi, vi
            mi, vi = self.calculate_mi_vi(XTrainData, YTestData,
XTrainData_label, Cn, alpha, beta,s,kernelType)

            NLL = gp.calculateMNLL(mi, vi, YTestData_label)
            # print('iteration: ' + str(iterationValues[i]) + ' ,alpha: ' +
str(alphaList[i]) + ', beta: ' + str(
            #     betaList[i]) + ' and MNLL: ' + str(np.mean(NLL)))
            MNLL_List.append(np.mean(NLL))

        # calculating MSE
        MSE = self.calculateMSE(mi, YTestData_label)

        return MSE, MNLL_List

    def evaluation_RBF(self, alphaList, betaList, sList, iterationValues,
XTrainData, YTestData, XTrainData_label,
                       YTestData_label,kernelType):
        MNLL_List = []
        for i in range(len(alphaList)):
            alpha = alphaList[i]
            beta = betaList[i]
            s = sList[i]

            # calculate Cn matrix for RBF kernel to calculate mi and vi
            # first recalculate RBF K Matrix with updated s
            K_RBF = self.createRBF_KMatrix(XTrainData, s)
            Cn = self.calculateCnMatrix(K_RBF, alpha, beta)

            # calculating mi, vi
            mi, vi = self.calculate_mi_vi(XTrainData, YTestData,
XTrainData_label, Cn, alpha, beta,s,kernelType)

            NLL = gp.calculateMNLL(mi, vi, YTestData_label)
            # print('iteration: ' + str(iterationValues[i]) + ' ,alpha: ' +
str(alphaList[i]) + ', beta: ' + str(
```

```
            #     betaList[i]) + ' and MNLL: ' + str(np.mean(NLL)))
            MNLL_List.append(np.mean(NLL))

        # calculating MSE
        MSE = self.calculateMSE(mi, YTestData_label)

        return MSE, MNLL_List

    def visualise1D(self, alpha_1D, beta_1D, s_1D, kernelType):
        # read train data
        # print('reading X')
        X = np.array(self.readData(train_1D))
        X = X.astype(np.float64)
        # print('X shape: ', X.shape)

        # train labels t
        X_t = np.array(self.readData(trainR_1D))
        X_t = X_t.astype(np.float64)
        # print('X_t shape: ', X_t.shape)

        # read test data
        Y = np.array(self.readData(test_1D))
        Y = Y.astype(np.float64)
        # print('Y shape: ', Y.shape)

        # test labels ti
        Y_ti = np.array(self.readData(testR_1D))
        Y_ti = Y_ti.astype(np.float64)

        if kernelType == 'Linear':
            # Linear K matrix
            K_1D = self.createKMatrix(X)

            # Cn matrix for linear
            Cn_1D = self.calculateCnMatrix(K_1D, alpha_1D, beta_1D)

            # obtain final values of alpha, beta and iterations after
gradient ascent
            alpha_1D, beta_1D, iterations, alphaList, betaList,
iterationValues = self.modelSelection_Linear(K_1D, X_t,

alpha_1D,

beta_1D,

eta, X)
            mi_1D, vi_1D = self.calculate_mi_vi(X, Y, X_t, Cn_1D, alpha_1D,
beta_1D, s_1D, 'Linear')

        else:
            # calculate RBF Kernel with updated s value
            K_1D = self.createRBF_KMatrix(X, s)

            Cn_1D = self.calculateCnMatrix(K_1D, alpha_1D, beta_1D)

            # # obtain final values of alpha, beta, s and iterations after
gradient ascent
```

```python
            alpha_1D, beta_1D, s_1D, iterations, alphaList, betaList, sList,
iterationValues = self.modelSelection_RBF(
                K_1D,
                X_t,
                alpha_1D,
                beta_1D,
                s_1D,
                eta,
                X)
            mi_1D, vi_1D = self.calculate_mi_vi(X, Y, X_t, Cn_1D, alpha_1D,
beta_1D, s_1D, 'RBF')

        fx_List = []

        for i in range(len(Y)):
            if Y[i] > 1.5:
                fx = -1
            elif Y[i] < -1.5:
                fx = 1
            else:
                fx = np.sin(6 * Y[i])
            fx_List.append(fx)

        stdDev = []

        for i in range(len(vi_1D)):
            stdDev.append(2*math.sqrt(vi_1D[i]))



        plt.plot(Y, fx_List, 'r', label='True Func')
        plt.plot(Y, mi_1D, 'b', label='Mean')
        plt.errorbar(Y,mi_1D,stdDev)
        plt.xlabel("x")
        plt.ylabel("f(x)")
        plt.title("Visualisation of 1D ")
        plt.legend()
        plt.show()


# initialising an object of class
gp = GP()

# read train data
XTrainData = np.array(gp.readData(train_artsmall))
XTrainData = XTrainData.astype(np.float64)

# train labels t
XTrainData_label = np.array(gp.readData(trainR_artsmall))
XTrainData_label = XTrainData_label.astype(np.float64)

# read test data
YTestData = np.array(gp.readData(test_artsmall))
YTestData = YTestData.astype(np.float64)

# test labels ti
YTestData_label = np.array(gp.readData(testR_artsmall))
```

```
YTestData_label = YTestData_label.astype(np.float64)

# hyper parameters
alpha = 1
beta = 1
s = 5

# eta
eta = 0.01

# Linear K matrix
K_Linear = gp.createKMatrix(XTrainData)

# RBF K Matrix
K_RBF = gp.createRBF_KMatrix(XTrainData, s)

#===========================================GP WITH LINEAR
KERNEL=======================================================
# obtain final values of alpha, beta and iterations after gradient ascent
alpha, beta, iterations, alphaList, betaList,iterationValues =
gp.modelSelection_Linear(K_Linear, XTrainData_label, alpha, beta, eta,
XTrainData)
print('LINEAR KERNEL: Final value of alpha: ' + str(alpha) + ', beta: ' +
str(beta) + ' and no of iterations: ' + str(
    iterations + 1))

# EVALUATION

# calculating MNLL at every 10th iteration
MSE,MNLL_List =
gp.evaluation_Linear(alphaList,betaList,iterationValues,K_Linear,XTrainData,Y
TestData,XTrainData_label,YTestData_label,'Linear')

print('MSE:',MSE)

plt.plot(iterationValues,MNLL_List)
plt.xlabel("iteration values")
plt.ylabel("MNLL")
plt.title("MNLL vs No. of iterations ")
plt.show()


# # # ========================================GP WITH RBF
KERNEL=============================================================
# # obtain final values of alpha, beta, s and iterations after gradient
ascent
# alpha, beta, s, iterations,alphaList,betaList,sList,iterationValues =
gp.modelSelection_RBF(K_RBF,XTrainData_label,alpha,beta,s,eta,XTrainData)
#
# print('RBF KERNEL: Final value of alpha: ' + str(alpha) + ', beta: ' +
str(beta) + ', s: ' + str(
#     s) + ' and no of iterations: ' + str(iterations + 1))
# #
# ##EVALUATION
#
# # calculating MNLL at every 10th iteration
# MSE, MNLL_List =
```

```
gp.evaluation_RBF(alphaList,betaList,sList,iterationValues,XTrainData,YTestDa
ta,XTrainData_label, YTestData_label,'RBF')
#
# print('MSE:', MSE)
#
# plt.plot(iterationValues,MNLL_List)
# plt.xlabel("iteration values")
# plt.ylabel("MNLL")
# plt.title("MNLL vs No. of iterations ")
# plt.show()


# =============================================VISUALISATION OF 1D
PERFORMANCE=============================================
# alpha_1D = 1
# beta_1D = 1
# s_1D = 0.1
#
# # Linear
# gp.visualise1D(alpha_1D,beta_1D,s_1D,'Linear')
#
# # RBF
# gp.visualise1D(alpha_1D,beta_1D,s_1D,'RBF')
```