Assignment V: Gaussian Process Regression

Exercises in Machine Learning (190.013), SS2022
Stefan Nehl[1]

[1]*stefan-christopher.nehl@stud.unileoben.ac.at, MNr: 00935188*, Montanuniversität Leoben, Austria

May 28, 2022

In the fifth assignment, I had to describe the Guassian Process Regression and implement it with the library GPy. Furthermore, I had to test different kernel implementations and hyper parameters and verify and compare the results with the results of the last assignment.

1 Gaussian Process Regression

Gaussian processes are a class of nonparametric models for machine learning. They are commonly used for modeling spatial and time series data. (Powell, 2021) The *Gaussian Process Regression* uses the *Multivariate Conditional Distribution*.

$$f(\boldsymbol{x'}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^{\mathrm{k}}|\boldsymbol{\Sigma}|}} exp^{-\frac{1}{2}(\boldsymbol{x'} - \boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{\text{-}1}(\boldsymbol{x'} - \boldsymbol{\mu})}$$

With $\boldsymbol{x'}=$ {x'$_1$, ..., x'_k}, $\boldsymbol{\Sigma}$ the covariance matrix and $|\boldsymbol{\Sigma}|$ the determinant of the covariance matrix. The covariance is determined by the covariance function of the kernel and has to be positive definite. (Rueckert, 2022) I tried three different kernels for the *Gaussian Process Regression*. The *Linear Kernel*, the *RBF, Radial Basis Function* and the *Matern 52*.

1.1 Linear Kernel

The *Linear Kernel* is based on linear classification. This classification is based on the linear combination of the characteristics. The decision function can be described with:

$$d(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) + b$$

where $\boldsymbol{w}$ is the weight vector, b a biased value and $\phi(\boldsymbol{x})$ a higher dimensional vector of **x**. If $\boldsymbol{w}$ is a linear combination of training data $\boldsymbol{w}$ can be calculated with:

$$\boldsymbol{w} = \sum_{i=1}^{l} \boldsymbol{\alpha}_i\phi(\boldsymbol{x_i})$$

for some $\boldsymbol{\alpha} \in \mathbf{R}^1$ The kernel function can be calculated with

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i))^T\phi(\boldsymbol{x}_j))$$

(Guo-Xun Yuan and Lin, 2021) The linear kernel is good for data with a lot of features. That's because mapping the data to a higher dimensional space does not really improve the performance. (KOWALCZYK, 2014) The implementation in the *GPy* library was the following:

$$K(x, y) = \sum_{i=1}^{D} \sigma_i^2 x_i y_i$$

Where D defines the dimension and σ_i^2 the variance for each dimension.

1.2 RBF

The *Radial Basis Function* is one of the most used kernels. It's similar to the *Gaussian distribution*. The kernel calculates the similarity or how close two points are to each other.

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \, epx(-\frac{||\boldsymbol{x}_i - \boldsymbol{x}_j||}{2\sigma^2})$$

Where $||\boldsymbol{x}_i - \boldsymbol{x}_j||$ is the euclidean (L_2-Norm) and σ the variance and the hyper parameter. (Sreenivasa, 2020)

$$K(r) = \sigma^2 \exp\left(-\frac{1}{2}r^2\right)$$

Implementation in the *GPy* library where $r = |\boldsymbol{x}_i - \boldsymbol{x}_j|$ and σ^2 the variance.

1.3 Matern 52

The *Matern 52* is a generalization of the *RBF* kernel.

$$K(r) = (1 + \frac{\sqrt{5}r}{l} + \frac{5r^2}{3l^2})\, exp(-\frac{\sqrt{5}r}{l})$$

Where $r = |\boldsymbol{x}_i - \boldsymbol{x}_j|$ and l a positive parameter.(C. E. Rasmussen, 2006) However, if I take a look in the code of the *GPy* implementation. It looks a little bit different.

$$K(r) = \sigma^2(1 + \sqrt{5}r + \frac{5}{3}r^2)\exp(-\sqrt{5}r)$$

It looks like, the positive parameter l is set to 1. As additional hyper parameter the variance, σ^2, was introduced.

1.4 Hyper-Parameters

The *GPy* library has for every kernel a variance parameter. This parameter is a hyper parameter to adjust improve the prediction result. If I set the parameter to a lower value the values for the learning are in a smaller gap. If I increase σ^2 the gab increases. So if the variance of my data is high, an increased value for the parameter variance makes sense.

1.5 Implementation

For the implementation of the *Gaussian Process Regression* i created a the class *GaussianProcess* which derived from the class *Regression* and reused the functions, *importData*, *generateTrainingSubset*, *createFeatureVector*, *testModel*, *computMeanOfError*, *getMeanError*, *plotError* and *plotHeatMap* from the last assignment about *Ridge Regression*. I implemented than the function *computeGaussianProcessRegression* which takes the parameters *kernelSetting* and *variance*. The parameter *kernelSetting* is an enum with the following values: I checked those values with an if and set the kernel to the corresponding value. The kernel function, *GPY.kern.KernelName* got 2 more parameters. One was the dimension of the data and the other one the variance, which is the hyper parameter. The dimension was set to two and for the variance I tried different values. After the kernel I created the model with the function *GPY.models.GPRegression* which takes the x and y values and the kernel as a parameter. The y values where normalized to have a mean of zero in the data and I used only a subset of the training data to overcome the performance issues. The size of the subset can be set with the parameter *trainStep* which has the same implementation from assignment 4. After the model creation I optimized the model with the *model.optimize* function. This function takes the max iterations as a parameter. I set this value to 1000. After the optimization I used the *model.predict* function and passed the y test data to the trained model. With the returned y values and the y test data from our data set, I calculated the error and the mean of the error. The last step was the plotting of the descending error, the heat map and compared the error with the error of the*Ridge Regression*.

Table 1: *Values for the kernel settings*

Value	String
LinearKernel	Linear Kernel
RBFWithGpu	RBF with GPU
RBF	RBF
Matern52	Matern 52

1.6 Result

First I tested which kernel performs the best. For this I created a test with the train steps of 20 and a hyper parameter of 1 and tested all three kernels and compared the error with the error of the *Ridge Regression*. Figure 1 displays the different values. (I had an issue in the last assignment with the *Ridge Regression*. This bug is now solved.) Figure 1 shows, that in this case the *Ridge Regression* performs better than the *Gaussian Processes* . From the *Gaussian Processes* the *Linear Kernel* performs better than the *Matern 52* and the *RBF*. Similar picture is also displayed in Figure 2 where I repeated the test, but with a variance of two. Also the optimization process took some time. Table 2 displays the optimization times from the different kernels. The times are not exact and are depending on the pc and if there are any other programs running in the background. Next I tested the different kernels with 4 different values for the variance. The values were 0.1, 0.5, 1 and 2. Figure

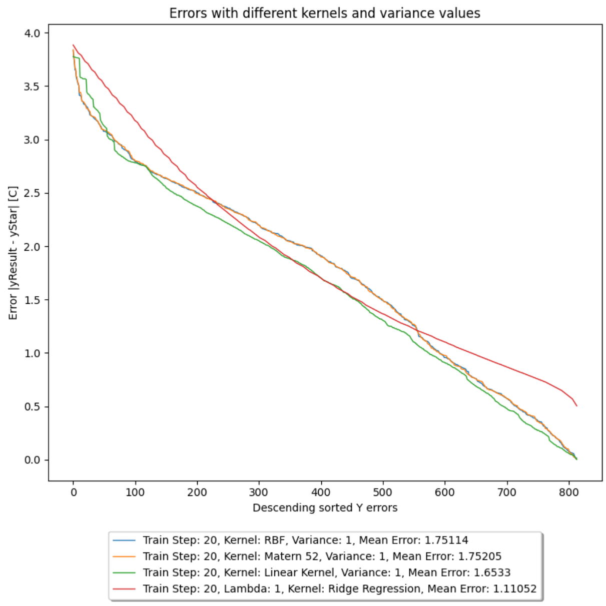

Figure 1: *Error with different kernels and variance:1*

Table 2: *Optimization time for the different kernels*

Kernel	Opt Time
RBF	1m 13
Matern 52	1m 29s
Linear Kernel	1m 4s

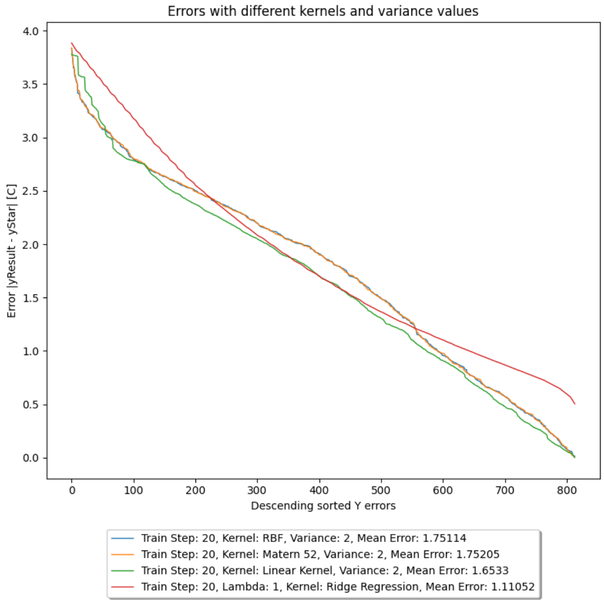

Figure 2: *Error with different kernels and variance:2*

3 shows the result of this test. The result displayed in

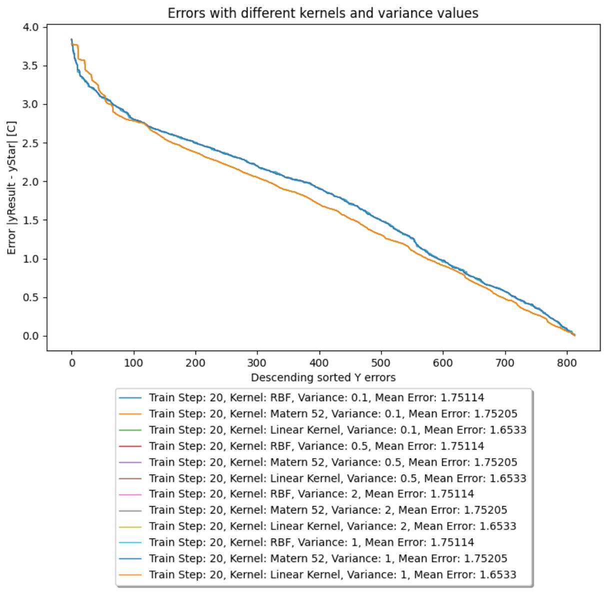

Figure 3: *Error with different kernels and variance:0.1, 0.5, 1, 2*

Figure 3 shows, that with a high amount of optimization iterations the variance effect is small or not even visible. I repeated the test without any optimization. Figure 4 shows that without optimization the mean error decreases with a lower variance for this dataset. This behaviour is also displayed in Figure 5 where I tested the *Matern 52* kernel with the different settings. Table 3 displays all results. Without the optimization there is no time measured for the optimization time. For the last test I used the *Linear Kernel* and compared the results with the *Ridge Regression*. For this I set the

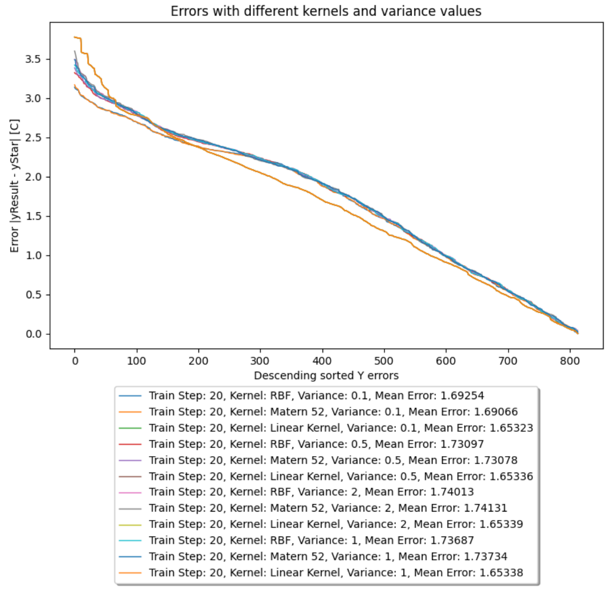

Figure 4: *Error with different kernels and variance:0.1, 0.5, 1, 2 and without optimization*

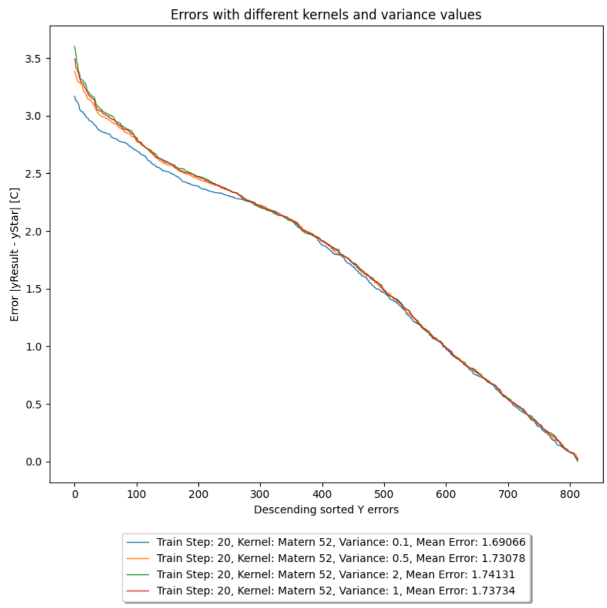

Figure 5: *Error with Matern 52 kernel and variance:0.1, 0.5, 1, 2 and without optimization*

Table 3: *Kernel Results*

Kernel	Variance	Optimized	Error Mean	Opt '
RBF	0.1	True	1.7511	1m
RBF	0.5	True	1.7511	1m
RBF	1.0	True	1.7511	1m
RBF	2.0	True	1.7511	1m
Matern 52	0.1	True	1.7520	1m
Matern 52	0.5	True	1.7520	1m
Matern 52	1.0	True	1.7520	1m
Matern 52	2.0	True	1.7520	1m
LK	0.1	True	1.6533	5(
LK	0.5	True	1.6533	1m
LK	1.0	True	1.6533	1m
LK	2.0	True	1.6533	1m
RBF	0.1	False	1.6925	·
RBF	0.5	False	1.7309	·
RBF	1.0	False	1.7368	·
RBF	2.0	False	1.7401	·
Matern 52	0.1	False	1.6906	-
Matern 52	0.5	False	1.7307	-
Matern 52	1.0	False	1.7373	-
Matern 52	2.0	False	1.7413	-
LK	0.1	False	1.6532	·
LK	0.5	False	1.6533	·
LK	1.0	False	1.6533	·
LK	2.0	False	1.6533	·

optimization value to 1000, the variance to 1 and the lambda value of the *Ridge Regression* to 1. I compared the error and the heat map of those regression models. Figure 6 displays the result. The mean of the *Ridge Regression* is smaller. However, the *Linear Kernel* performs better in some areas. This is also visible in Figure 7 and 8 which displays the heat map of both. The *Linear Kernel* performs better in the center of the given longitude and latitude dataset and the *Ridge Regression* better and the right side of the dataset.

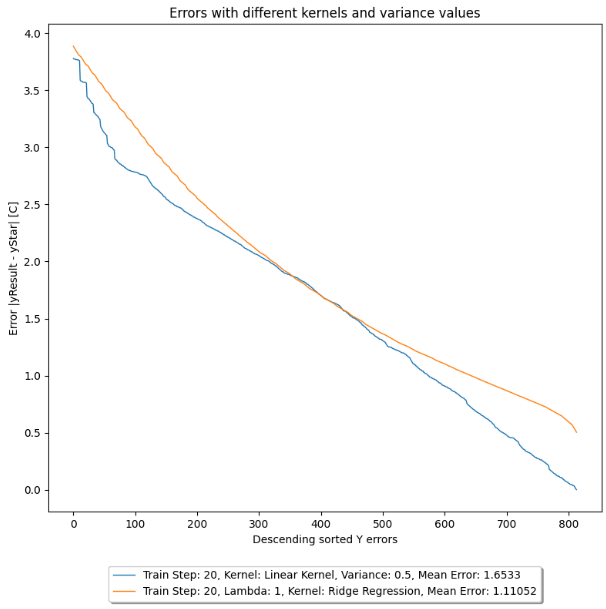

Figure 6: *Error with Ridge Regression and Linear Kernel*

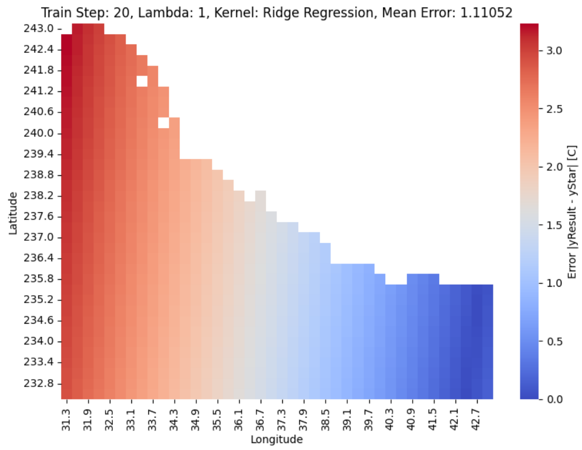

Figure 7: *Heatmap Ridge Regression*

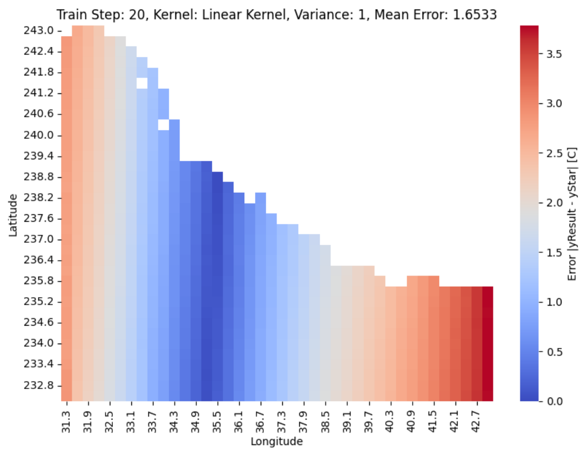

Figure 8: *Heatmap Linear Kernel*

1.7 Conclusion

The implementation with the *GPy* was easy and with the help of the tutorials good doable. The optimization of the kernel has a big performance impact on the mean and does not always provide better results. I mixture between smaller optimization values and a good picked hyper parameter can improve the overall results. However, in all the cases the *Ridge Regression* had a better mean than the *Gaussian Processes*.

Bibliography

C. E. Rasmussen, C. K. I. Williams (2006). *Gaussian Processes for Machine Learning*. MIT Press, p. 85.

Guo-Xun Yuan, Chia-Hua Ho and Chih-Jen Lin (2021). "Recent Advances of Large-Scale Linear Classification". In: *Proceedings of the IEEE* 100, pp. 2584–2603.

KOWALCZYK, Alexandre (2014). *Linear Kernel: Why is it recommended for text classification*. URL: `https://www.svm-tutorial.com/2014/10/svm-linear-kernel-good-text-classification/`.

Powell, Alex (2021). *Multivariate normal distribution*. URL: `https://towardsdatascience.com/intro-to-gaussian-process-regression-14f7c647d74d`.

Rueckert, Elmar (2022). *An Introduction to Probabilistic Machine Learning*. Elmar Rueckert.

Sreenivasa, Sushanth (2020). *Radial Basis Function (RBF) Kernel: The Go-To Kernel*. URL: `https://www.svm-tutorial.com/2014/10/svm-linear-kernel-good-text-classification/`.

APPENDIX

```python
import sys
import seaborn as sbr
import matplotlib.pyplot as plt

sys.path.insert(0, '../modules')
from RidgeRegression import RidgeRegression
from GaussianProcess import GaussianProcess
from GaussianProcess import KernelSetting

def doTestGauss(trainStep, kernelSetting:KernelSetting, variance:float, plot=False):
    gausRegression = GaussianProcess(trainStep)
    print("import data")
    gausRegression.importData()
    print("generate Trainingset")
    gausRegression.generateTrainingSubset()
    print("Train")
    gausRegression.computeGaussianProcessRegression(kernelSetting, variance=variance)
    gausRegression.testModel()
    print("Calculate Error")
    yTest = gausRegression.getYTestData()
    gausRegression.computeError(yTest)
    gausRegression.computMeanOfError()
    print("Mean Error with kernel: " + kernelSetting.value + " variance " + str(variance) + " : " + str(gausRegression.getMeanError()))

    if plot:
        print("plot error")
        gausRegression.plotError()
        print("plot heatmap")
        gausRegression.plotHeatMap()

    return gausRegression.getDescSortedError(), gausRegression.getSettingsString()

def doTestRidge(trainStep, lambdaValue, plot=False):
    ridgeRegression = RidgeRegression(trainStep)
    print("import data")
    ridgeRegression.importData()
    print("generate Trainingset")
    ridgeRegression.generateTrainingSubset()
    print("Train")
    weightVector = ridgeRegression.computeLinearRidgeRegression(lambdaValue)
    ridgeRegression.testModel(weightVector)
    print("Calculate Error")
    yTest = ridgeRegression.getYTestData()
    ridgeRegression.computeError(yTest)
    ridgeRegression.computMeanOfError()
    print("Mean Error with Lambda " + str(lambdaValue) + ": " + str(ridgeRegression.getMeanError()))

    if plot:
        print("plot error")
        ridgeRegression.plotError()
        print("plot heatmap")
        ridgeRegression.plotHeatMap()

```

```
    return ridgeRegression.getDescSortedError(), ridgeRegression.
        getSettingsString()

firstTrainSetErrors = []

trainStep = 20
variances = {0.1}

for i in variances:
    #firstTrainSetErrors.append(doTestGauss(trainStep,variance=i, kernelSetting=
        KernelSetting.RBF))
    #firstTrainSetErrors.append(doTestGauss(trainStep, variance=i, kernelSetting
        =KernelSetting.Matern52))
    firstTrainSetErrors.append(doTestGauss(trainStep, variance=i, kernelSetting=
        KernelSetting.LinearKernel, plot=True))

lambdaValues = {1}
for i in lambdaValues:
    firstTrainSetErrors.append(doTestRidge(trainStep, i, plot=True))

plt.figure(figsize=(8, 8))
for testSet in firstTrainSetErrors:
    errors = testSet[0]
    settings = testSet[1]
    x = errors["values"]
    y = errors["error"]
    plt.plot(x, y, linewidth=1, label=settings)

plt.title("Errors with different kernels and variance values")
plt.xlabel("Descending sorted Y errors")
plt.ylabel("Error |yResult - yStar| [C]")
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.10),
          fancybox=True, shadow=True, ncol=1)
plt.tight_layout()
plt.show()
```

```python
import matplotlib.pyplot
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import seaborn as sbr
import pandas as pd
import GPy

from inference import Regression
from BasicStatistics import BasicStatistics
from GaussDistribution import GaussDistribution
from enum import Enum

class KernelSetting(Enum):
    LinearKernel = "Linear Kernel"
    RBFWithGpu = "RBF with GPU"
    RBF = "RBF"
    Matern52 = "Matern 52"

class GaussianProcess(Regression):

    def __init__(self, trainStep):
        self.trainStep = trainStep

    def importData(self):
        dataDictonary =  sio.loadmat("AssignmentIV_data_set.mat")

        tempTimeData = dataDictonary.get("TempField")
        self.tempData = np.array(tempTimeData[:,:, 0])

        longData = dataDictonary.get("LongitudeScale") #x-value: Long
        latData = dataDictonary.get("LatitudeScale") #y-value: Lat

        self.x_test = np.array(dataDictonary.get("x_test")) # Testdata from
            DataSet

        normalizedY_TestValues = np.matrix(BasicStatistics(dataDictonary.get("
            y_test")[0]).getNormalizeDataSet())
        self.y_test = np.transpose(np.array(normalizedY_TestValues)) # Testdata
            from DataSet

        longLatData = []
        arrayValues = []

        for y in range(len(self.tempData)):
            for x in range(len(self.tempData[y])):
                value = self.tempData[y,x]
                if np.isnan(value):
                    continue

                arrayValues.append(value)
                longLatData.append((latData[y][0], longData[x][0])) #[0] needed
                    because of strange import of long/latData

        self.inputValues = longLatData

        normalizedYValues =  BasicStatistics(arrayValues)
        self.outputValues = np.array(normalizedYValues.getNormalizeDataSet())
```

```
        self.numberOfSamples = len(self.inputValues)

    def generateTrainingSubset(self):
        self.trainSubsetInput = np.array(self.inputValues[0:len(self.inputValues
            ):self.trainStep])
        self.trainSubsetOutput = np.array(self.outputValues[0:len(self.
            outputValues):self.trainStep])

    def createFeatureVector(self, x):
        featureVector = []

        for i in range(len(x)):
            newXVector = x[i]
            featureVector.append(newXVector)

        return featureVector

    def computeGaussianProcessRegression(self, kernelSetting:KernelSetting,
       variance:float):
        X = np.vstack(([self.createFeatureVector(x) for x in self.
            trainSubsetInput]))
        Y = np.vstack(([y for y in self.trainSubsetOutput]))


        self.kernelSetting = kernelSetting
        self.ard = False
        self.iterations = 1000
        self.variance = variance

        kernel = object
        if self.kernelSetting == KernelSetting.LinearKernel:
            kernel = GPy.kern.Linear(2, variances=variance, ARD=self.ard)

        if self.kernelSetting == KernelSetting.RBF:
            kernel = GPy.kern.RBF(2, variance=variance, ARD=self.ard, useGPU=
                False)

        if self.kernelSetting == KernelSetting.RBFWithGpu:
            kernel = GPy.kern.RBF(2, variance=variance, ARD=self.ard, useGPU=
                True)

        if self.kernelSetting == KernelSetting.Matern52:
            kernel = GPy.kern.Matern52(2, variance=variance, ARD=self.ard)

        self.model = GPy.models.GPRegression(X=X, Y=Y, kernel=kernel)
        self.model.optimize(messages=True, max_iters=self.iterations)

        return self.model

    def testModel(self):
        xNew = self.x_test[0:,1:]
        result = self.model.predict(xNew)
        self.yResult = np.array(result[0])


    def getYTestData(self):
        return self.y_test

    def computeError(self, yStar):
```

```python
        self.yError = abs(self.yResult - yStar)
        reversedArray = np.flip(np.sort(self.yError, 0))
        self.errorDataFrame = pd.DataFrame({
            'values': range(len(reversedArray)),
            'error': [error[0] for error in reversedArray]
        })

    def getDescSortedError(self):
        return self.errorDataFrame

    def computMeanOfError(self):
        self.meanError = np.mean(self.yError)
        self.settingsString = (f"Train Step: {self.trainStep}, "
                               f"Kernel: {self.kernelSetting.value}, "
                               f"Variance: {self.variance}, "
                               f"Mean Error: {round(self.meanError, 5)}")
    def getSettingsString(self):
        return self.settingsString

    def getMeanError(self):
        return self.meanError

    def plotError(self):
        plt.figure(figsize=(8, 6))
        errorPlot = sbr.barplot(data=self.errorDataFrame, x="values", y="error",
             palette="coolwarm_r")

        plt.xlabel("Descending Sorted Y Errors")
        plt.ylabel("Error |yResult - yStar| [C]")
        plt.title(self.settingsString)
        plt.tight_layout()
        matplotlib.pyplot.show()

    def plotHeatMap(self):
        tempErrorData = \
            {
                'Lat': [round(long,2) for long in self.x_test[:, 1]],
                'Long': [round(lat,2) for lat in self.x_test[:, 2]],
                'error':[error[0] for error in self.yError]
            }

        tempDataFrame = pd.DataFrame(tempErrorData)
        tempDataFrame = tempDataFrame.pivot("Long", "Lat", "error")
        reversedTempErrorData = tempDataFrame.sort_values(("Long"), ascending=
            False)

        def fmt(x, y):
            return '{:,.2f}'.format(x)

        plt.figure(figsize=(8,6))
        errorHeatMap = sbr.heatmap(reversedTempErrorData, vmin=0.0, cmap="
            coolwarm", cbar_kws={"label":"Error |yResult - yStar| [C]"})
        ax = errorHeatMap.axes

        plt.xlabel("Longitude")
        plt.ylabel("Latitude")
        plt.title(self.settingsString)
        plt.tight_layout()
        matplotlib.pyplot.show()
```

```python
import csv
import numpy as np
import matplotlib.pyplot as plt
import math

from abc import ABC, abstractmethod

class ContiniousDustribution():

    @abstractmethod
    def importCsv(self, filename):
        pass

    @abstractmethod
    def exportCsv(self, filename):
        pass

    @abstractmethod
    def calculateMean(self):
        pass

    @abstractmethod
    def calculateVariance(self):
        pass

    @abstractmethod
    def calculateStandardDeviation(self):
        pass

    @abstractmethod
    def normalizeDataSet(self):
        pass

    @abstractmethod
    def generateSampels(self):
        pass

    @abstractmethod
    def plotData(self):
        pass

class Regression():

    @abstractmethod
    def importData(self):
        pass

    @abstractmethod
    def generateTrainingSubset(self):
        pass

    @abstractmethod
    def computeLinearRidgeRegression(self, lambdaValue):
        pass

    @abstractmethod
    def testModel(self, weight):
        pass

```

```
    @abstractmethod
    def computeError(self, yStar):
        pass

    @abstractmethod
    def plotError(self):
        pass

    @abstractmethod
    def plotHeatMap(self):
        pass

    @abstractmethod
    def computMeanOfError(self):
        pass
```