PRINCIPLES OF MACHINE LEARNING

Exercise Sheet 3

30.11.2023

Richard Restetzki	2973740
Akmalkhon Khashimov	50178353
Valdrin Smakaj	50138041
Suraj Giri	50190564
Nijat Sadikhov	50186266
Nikita Morev	50134788
Aleksandr Semenikhin	50118777
Vibhor Sharma	50010826
Muslimbek Abduvaliev	50136555
Suyash Thapa	50205756
Mohammad Mehdi Deylamipour	50009389
Nicolás López Funes	50151598
Andrii Shevliakov	50198078
Nurmukhammad Aberkulov	50102727

Task 3.1

Basic functionalities (implement diffMatrix and prodMatrix functions)

```
 # takes two inputs u and v and returns their difference as a matrix
def diffMatrix(u, v):
    u = np.array(u)[:, None]
    v = np.array(v)[None, :]
    return u – v


#  takes two inputs u and v and returns their product as a matrix

def prodMatrix(u, v):
    u = np.array(u)[:, None]
    v = np.array(v)[None, :]
    return u * v
```

Input:
u = [0 1 2]
V = [0 1 2 3 4 5]

Output:
DiffMatrix = [[0 -1 -2 -3 -4 -5]
[1 0 -1 -2 -3 -4]
[2 1 0 -1 -2 -3]]

ProdMatrix = [[0 0 0 0 0 0]
[0 1 2 3 4 5]
[0 2 4 6 8 10]]

Task 3.2

Kernel matrices

```
 # takes two inputs u and v and one parameter α and returns linear kernel matrix
def linearKernelMatrix(u, v, alpha):
    K = alpha * prodMatrix(u, v)
    return K


#  takes two inputs u and v and two parameters α and σ  and returns gaussian kernel matrix

def gaussianKernelMatrix(u, v, alpha, sigma):
    diffMatrix = diffMatrix(u, v)

    K = alpha * np.exp((-1) * (diffMatrix ** 2) / (2 * (sigma ** 2)))
    return K
```

Input:
u = [0 1 2]
V = [0 1 2 3 4 5]
alpha = 0.5
Sigma = 1

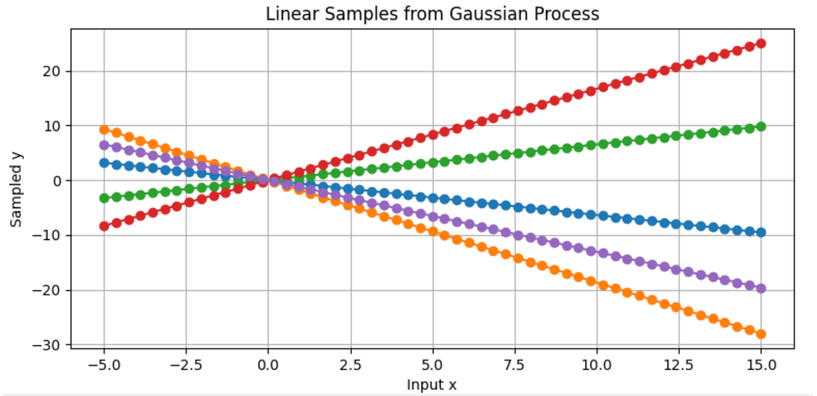
Output:

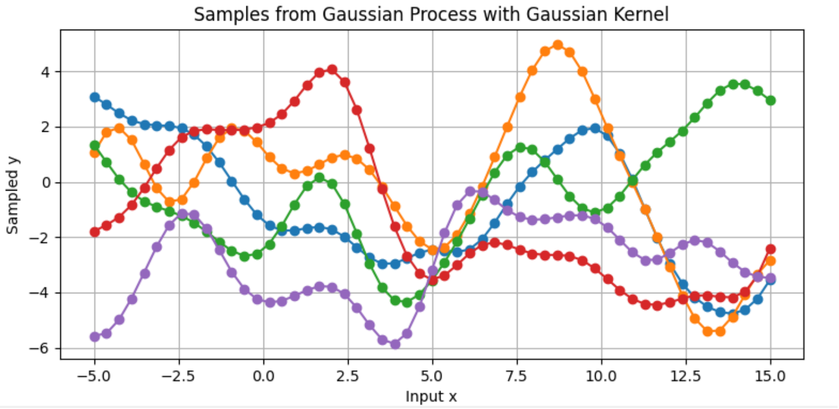
```
linearKernelMatrix = [[0.  0.  0.  0.  0.  0. ]
                      [0.  0.5 1.  1.5 2.  2.5]
                      [0.  1.  2.  3.  4.  5. ]]

ProdMatrix = [[5.00000000e-01  3.03265330e-01  6.76676416e-02  5.55449827e-03  1.67731314e-04  1.86332659e-06]
              [3.03265330e-01  5.00000000e-01  3.03265330e-01  6.76676416e-02  5.55449827e-03  1.67731314e-04]
              [6.76676416e-02  3.03265330e-01  5.00000000e-01  3.03265330e-01  6.76676416e-02  5.55449827e-03]]
```

Task 3.3

Sampling simple Gaussian processes

```
# number of samples to work with
num_samples = 5

# input vector and zero vector
vecX = np.linspace(-5.0, 15.0, 55)
vec0 = np.zeros_like(vecX)


# task 3.3.1
# Value for the kernel parameter
alphaL = 1.0

# Kernel matrix KL
KL = alphaL * np.outer(vecX, vecX)

# Sample 5 vectors from the Gaussian process using linear kernel matrix
samples_linear = [np.random.multivariate_normal(vec0, KL) for _ in range(num_samples)]



# task 3.3.2
# Parameters for Gaussian kernel
alpha_G = 6.0
sigma_G = 1.5

# Gaussian kernel matrix KG
KG = alpha_G * np.exp(-np.subtract.outer(vecX, vecX)**2 / (2 * sigma_G**2))

# Sample 5 vectors from the Gaussian process using the Gaussian kernel matrix
samples_gaussian = [np.random.multivariate_normal(vec0, KG) for _ in range(num_samples)]
```



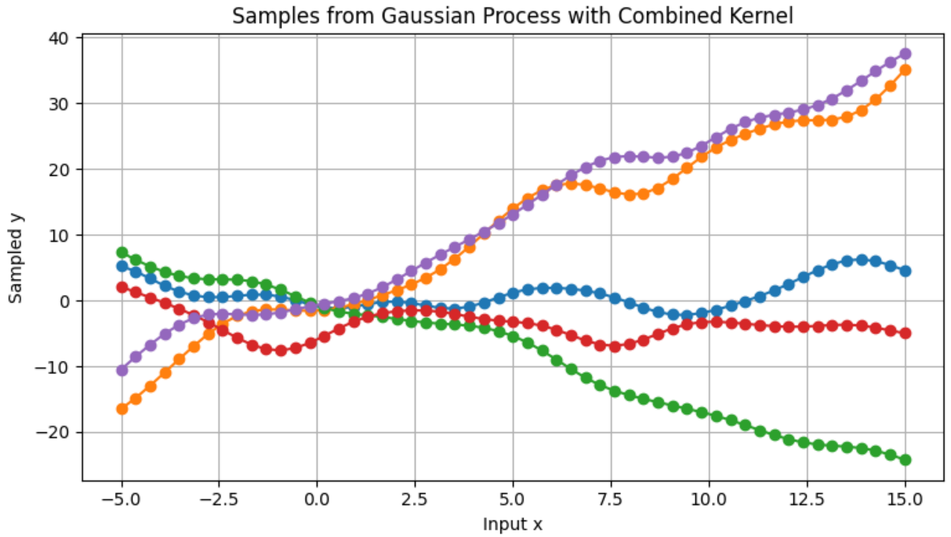


Task 3.3

Sampling simple Gaussian processes

```
# number of samples to work with
num_samples = 5

# input vector and zero vector
vecX = np.linspace(-5.0, 15.0, 55)
vec0 = np.zeros_like(vecX)
```

```
# task 3.3.3
#Kernel parameters for the combined kernel
alpha_L = 2.0
alpha_G = 6.0
sigma_G = 1.5

# Create linear kernel matrix KL and KG and combine them
KL = alpha_L * np.outer(vecX, vecX)
KG = alpha_G * np.exp(-np.subtract.outer(vecX, vecX)**2 / (2 * sigma_G**2))
KLG = KL + KG

# Sample 5 vectors from the Gaussian process using the combined kernel matrix
samples_combined = [np.random.multivariate_normal(vec0, KLG) for _ in range(num_samples)]
```



Task 3.4

Fitting a Gaussian processes model to data

```
# Normalize y
y_mean = np.mean(y)
y_normalized = y - y_mean

# kernel function
def kernel(x, theta):
    theta1, theta2, theta3, theta4 = theta
    n = len(x)
    K = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            K[i, j] = (theta1 * np.exp(-((x[i] - x[j])**2) / theta2**2) +
                       theta3 * x[i] * x[j] +
                       (1 if i == j else 0) * theta4)
    return K

# negative likelihood function
def negLikelihood(theta, x, y):
    C = kernel(x, theta) + theta[3] * np.eye(len(x))
    return 0.5 * np.log(np.linalg.det(C)) + 0.5 * np.dot(y, np.linalg.solve(C, y))

# Optimize parameters
initial_theta = [1.0, 20.0, 0.5, 1.0]
bounds = [(0, None), (0, None), (0, None), (0, None)]  # Non-negative parameters
result = optimize.minimize(negLikelihood, initial_theta, args=(x, y_normalized), bounds=bounds)
```

Output:

```
[31.37874554  14.35857048   0.        73.29727899]
```

We defined a kernel function and a negative likelihood function for optimizing the parameters of a Gaussian process.

The kernel function computes the covariance matrix based on an input vector x and a parameter vector θ, which includes terms for radial-basis function (RBF), linear, and noise components.

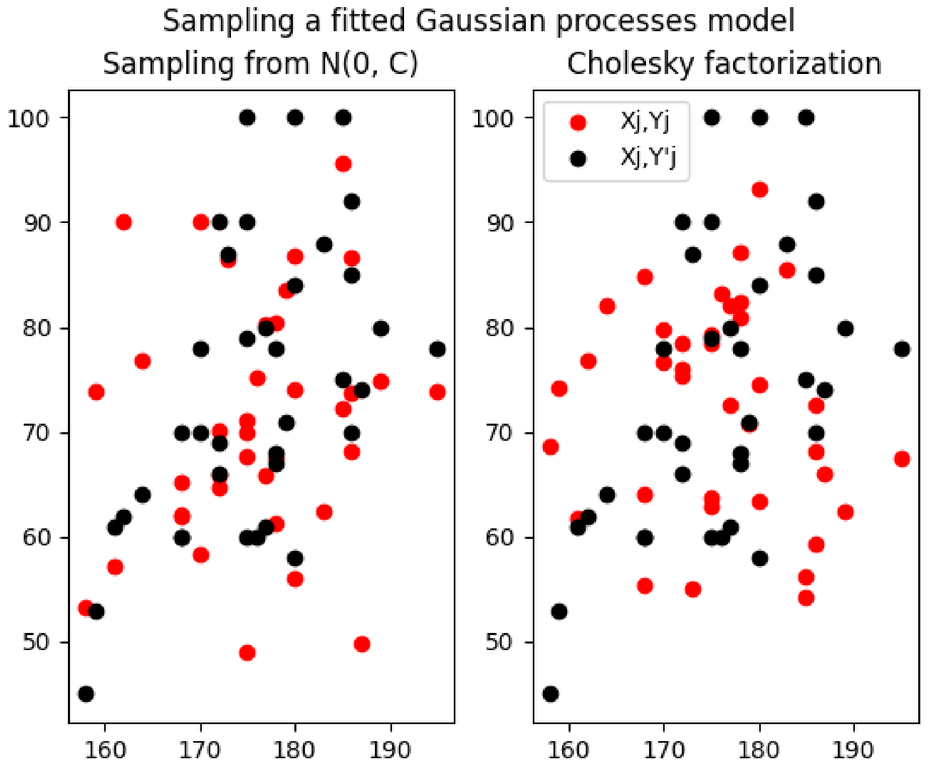
The negative likelihood function calculates the log-likelihood of the observed data y, given the input x and parameters θ, and is used as an objective function to minimize during the optimization process.

The optimization is performed using the optimize.minimize function from SciPy, with initial parameters and bounds set to ensure all parameters are non-negative.

Task 3.5

Sampling a fitted Gaussian processes model

```
def sample_from_gaussian(x, theta):
    C = kernel(x, theta) + theta[3] * np.eye(len(x))
    # Sample a vector
    y_prime = np.random.multivariate_normal(np.zeros(len(C)), C)
    # De-normalize y prime
    y_denormalized = y_prime + y_mean
    return y_denormalized


def cholesky_factorization(x, theta):
    C = kernel(x, theta) + theta[3] * np.eye(len(x))
    # Cholesky factor
    L = np.linalg.cholesky(C)
    w = np.random.multivariate_normal(np.zeros(len(x)), np.eye(len(x)))
    # Compute y prime
    y_prime = L @ w
    # De-normalize y prime
    y_denormalized = y_prime + y_mean
    return y_denormalized
```



The sample_from_gaussian function samples from a multivariate normal distribution defined by a Gaussian process, using a covariance matrix C generated by the kernel function with added noise from θ_3. It then re-centers the sampled data around the original data mean, y_mean, effectively de-normalizing the sample.

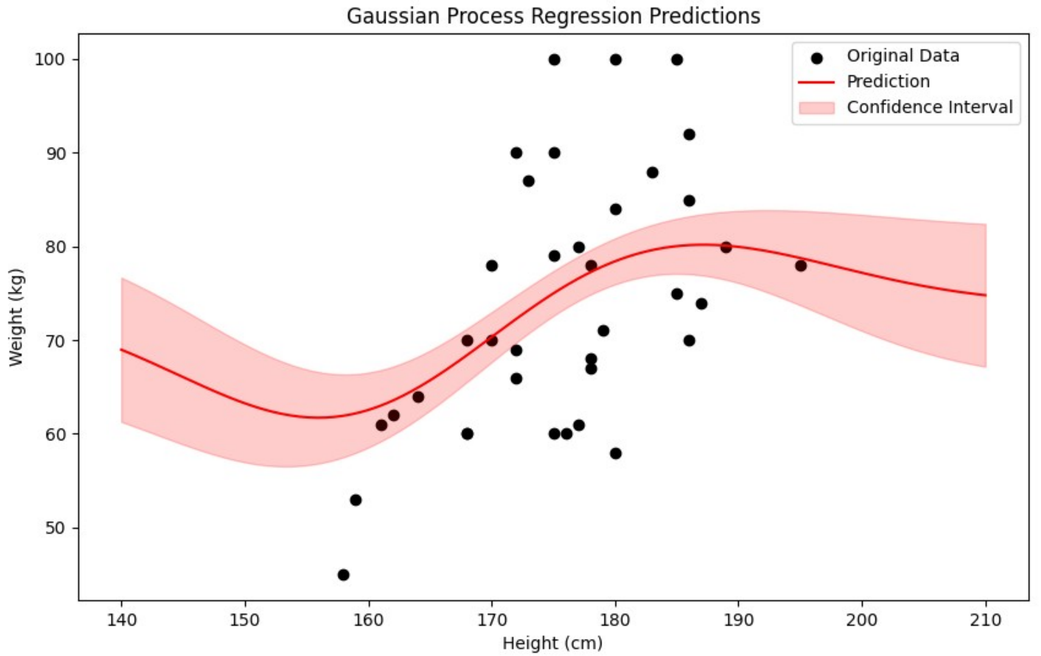
The cholesky_factorization function performs a similar task but uses the Cholesky decomposition to sample more efficiently from the multivariate normal distribution. It obtains a lower triangular matrix L from the covariance matrix C, then samples a vector w from a standard normal distribution, which is transformed by L to obtain the sample y′. This sample is also de-normalized using the mean of the original data, y_mean.

Both methods generate samples from the Gaussian process but use different techniques for the sampling process.

Task 3.6

Predicting with a fitted Gaussian processes model

```
# New input values for prediction
x_star = np.linspace(140, 210, 200)

# Compute Kernel Matrices for predictions
K_xx = kernel(x, result.x[:3])
K_xstar_x = kernel(x_star, result.x[:3], x)
K_xxstar = kernel(x, result.x[:3], x_star)
K_xstar_xstar = kernel(x_star, result.x[:3])

# Covariance matrix C for predictions
C = K_xx + result.x[3] * np.eye(len(x))

# Predictions
mu_star = K_xstar_x @ np.linalg.inv(C) @ y_normalized
Sigma_star = K_xstar_xstar - K_xstar_x @ np.linalg.inv(C) @ K_xxstar
sigma_star = np.sqrt(np.diag(Sigma_star))

# De-normalize predictions
mu_star_denorm = mu_star + y_mean
```



In the provided code, x_star defines new input values for height, across a range, for which the weight predictions are to be made using a Gaussian process model. The predictive mean mu_star and covariance Sigma_star are calculated using kernel matrices based on the optimized hyperparameters found earlier (result from task 3.4), where mu_star captures the expected weights and Sigma_star measures the uncertainty of these predictions. Finally, the predictive mean mu_star is de-normalized by adding the original mean weight y_mean to yield predictions in the original scale of the data.