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

PRINCIPLES OF MACHINE LEARNING

Exercise Sheet 1

TASK 1 - NUMERICAL INSTABILITIES

Task 1.1.1

<u>Code:</u>

```
matX = np.array([[ 1.00000, 0.00000, 0.00000], [-1.00000, 0.00001, 0.00000]])

vecY = np.array( [ 0.00000, 0.00001, 0.00000] )

vecW = la.inv(matX @ matX.T) @ matX @ vecY

print (vecW)
```

<u>Output:</u> *[0.99999992 0.99999992]*

The solution is off by a slight bit because the matrix inverse cannot be computed exactly with the given precision (Matrix is ill-conditioned)

TASK 1 - NUMERICAL INSTABILITIES

Task 1.1.2

Code:

```
matX = np.array([[ 1.00000, 0.00000, 0.00000], [-1.00000, 0.00001, 0.00000]])

vecY = np.array( [ 0.00000, 0.00001, 0.00000] )

Xt_Q,Xt_R = la.qr(matX.T)

vecW = la.inv(Xt_R) @ Xt_Q.T @ vecY

print (vecW)
```

Output: *[1. 1.]*

The solution is accurate because we only need to invert an upper triange matrix which can be done exactly (Matrix is still ill-conditioned but our calculation doesn't propagate)

C2 General

TASK 1 - NUMERICAL INSTABILITIES

Task 1.1.3

Code:

```
matX = np.array([[ 1.00000, 0.00000, 0.00000], [-1.00000, 0.00001, 0.00000]])

vecY = np.array( [ 0.00000, 0.00001, 0.00000] )

vecW = la.lstsq(matX.T, vecY, rcond=-1)[0]

print (vecW)
```

Output: *[1. 1.]*

The method does not use the equation (3) $w_* = [XX^T]^{-1}Xy$ directly. As the parameter "rcond=-1" enforces the la.lstsq() function to use machine precision and we also used machine precision for Task 1.1.1 where (3) is used, the accurate Output proves that “la.lstsq()” does not use equation (3)

C2 General

TASK 1 - NUMERICAL INSTABILITIES

Task 1.1.4

- Rounding errors will emerge when using machine precision
- One should always think about the condition of the input arguments and the error propagation within our calculations
- In this case the posed problem with input-matrix matX is not very well-conditioned
- Inverting any matrix should always be avoided where possible. As seen above, the replacement of a matrix inversion by inverting an upper triangle matrix is sufficient in this case

TASK 2 - CELLULAR AUTOMATA

Task 1.2.1

Code:

```
matX = np.array([[ 1,  1,  1,  1, -1, -1, -1, -1], [ 1,  1, -1, -1,  1,  1, -1, -1], [ 1, -1,  1, -1,  1, -1,  1, -1]])

vecY110 = np.array([ 1,-1,-1,-1, 1,-1,-1, 1])

vecW110 = la.lstsq(matX.T, vecY110, rcond=None)[0]

vecYhat110 = matX.T @ vecW110
```

$X^\intercal$			y
$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	-1	-1
$+1$	-1	$+1$	-1
$+1$	-1	-1	-1
-1	$+1$	$+1$	$+1$
-1	$+1$	-1	-1
-1	-1	$+1$	-1
-1	-1	-1	$+1$

rule 110

$$\boldsymbol{w}_\star = \underset{\boldsymbol{w}\in\mathbb{R}^3}{\operatorname{argmin}} \left\| \boldsymbol{X}^\intercal \boldsymbol{w} - \boldsymbol{y} \right\|^2$$

$$\hat{\boldsymbol{y}} = \boldsymbol{X}^\intercal \boldsymbol{w}_\star$$

Output:

```
vecY110:  [ 1 -1 -1 -1  1 -1 -1  1] ; vecYhat110:  [ 0.25 -0.25 -0.25 -0.75  0.75  0.25  0.25 -0.25]

Residual for rule 110:  [ 0.75 -0.75 -0.75 -0.25  0.25 -1.25 -1.25  1.25]
```

The least squares problem cannot be fitted appropriately by linear models. In a way, we are prescribing a random 8-dimensional target vector to our 3-dimensional feature map and expecting the emerging problem to be of linear nature which it is unsurprisingly not

C2 General

TASK 2 - CELLULAR AUTOMATA

Task 1.2.2 - Theory

Let us denote our index set of the entries x_j of $\boldsymbol{x} \in \mathcal{B}^{\boldsymbol{n}}$ by $I := \{1,2,\ldots,n\}$

The Boolean Fourier series expansion is defined as:

$$f(\boldsymbol{x}) = \sum_{\mathcal{S} \in 2^{\mathcal{I}}} w_{\mathcal{S}} \prod_{j \in \mathcal{S}} x_j \quad \text{with} \quad \varphi_{\mathcal{S}}(\boldsymbol{x}) = \prod_{j \in \mathcal{S}} x_j$$

Therefore, we can rewrite: $$f(\boldsymbol{x}) = \sum_{\mathcal{S} \in 2^{\mathcal{I}}} w_{\mathcal{S}}\, \varphi_{\mathcal{S}}(\boldsymbol{x}) = \boldsymbol{w}^{\intercal} \boldsymbol{\varphi}(\boldsymbol{x})$$

<u>Task:</u> Implement $\boldsymbol{\varphi}(\boldsymbol{x})$

TASK 2 - CELLULAR AUTOMATA

Task 1.2.2 - Application

Code:

```
def powerset(iterable):
    # powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return iter.chain.from_iterable(iter.combinations(s, r) for r in range(len(s)+1))

def phi(vecX):
    # This function realizes the transformation phi by taking any vector x and returning the vector phi(x) with size 2^n
    n = vecX.shape[0]
    vecPhiX = np.zeros(pow(2,n))
    iteration = 0
    for S in powerset(range(n)):
        # This loop goes over every set S in the powerset of the index set I {0,1,...,n-1} (note that here the first index is 0 for obv. reasons)
        # The powerset of the index set is used instead of creating the powerset of {x_i} as the x_i's might get large leading to a high memory usage
        entry = 1
        for index in S:
            # This loop realizes the multiplication needed for computing phi_S(x)
            entry *= vecX[index]
        vecPhiX[iteration] = entry
        iteration += 1
    return vecPhiX
```

Result:

```
print (phi(np.array([2,3,5,7])))
```

 [1. 2. 3. 5. 7. 6. 10. 14. 15. 21. 35. 30. 42. 70. 105. 210.]

TASK 2 - CELLULAR AUTOMATA

Task 1.2.3 - Breakdown of Task

We can write our initial matrix X like this: $\boldsymbol{X}^\intercal = \begin{bmatrix} — \boldsymbol{x}_0^\intercal — \\ — \boldsymbol{x}_1^\intercal — \\ \vdots \\ — \boldsymbol{x}_7^\intercal — \end{bmatrix}$ where $\boldsymbol{x}_0 = \begin{bmatrix} +1 \\ +1 \\ +1 \end{bmatrix}$ $\boldsymbol{x}_1 = \begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix}$ $\dots$ $\boldsymbol{x}_7 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$

Now implement the following feature matrix: $\boldsymbol{\Phi}^\intercal = \begin{bmatrix} — \boldsymbol{\varphi}_0^\intercal — \\ — \boldsymbol{\varphi}_1^\intercal — \\ \vdots \\ — \boldsymbol{\varphi}_7^\intercal — \end{bmatrix}$ where $\boldsymbol{\varphi}_j = \varphi(\boldsymbol{x}_j)$

Then compute $\boldsymbol{w}_\star = \underset{\boldsymbol{w} \in \mathbb{R}^8}{\operatorname{argmin}} \left\| \boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y} \right\|^2$ and $\hat{\boldsymbol{y}} = \boldsymbol{\Phi}^\intercal \boldsymbol{w}_\star$

TASK 2 - CELLULAR AUTOMATA

Task 1.2.3 - Application

Code:

```
def Phi(matX):
    n, size = matX.shape
    matPhiXt = np.zeros( (size, pow(2,n)) )
    for column in range(size):
        # We replace each of the vectors x_i.T (rows of matX.T) with the respective lifted vectors phi_i.T (rows of matPhiXt)
        matPhiXt[column] = phi(matX.T[column])
    return matPhiXt.T

matPhiX = Phi(matX)

vecW110 = la.lstsq(matPhiX.T, vecY110, rcond=None)[0]
vecYhat110 = matPhiX.T @ vecW110
print("vecY110: ", vecY110, "; vecYhat110: ", vecYhat110)
print("Residual for rule 110: ", vecY110-vecYhat110)
```

Output:

```
vecY110:  [ 1 -1 -1 -1  1 -1 -1  1] ; vecYhat110:  [ 1. -1. -1. -1.  1. -1. -1.  1.]
Residual for rule 110:  [ 0.00000000e+00 -1.11022302e-16  2.22044605e-16 -3.33066907e-16 -4.44089210e-16
0.00000000e+00 -4.44089210e-16  2.22044605e-16]
```

Result: The residual is negligible (at machine precision level). We had to pay for the good fit by increasing the dimension of the feature space to 2^n (=8 in this case). This always allows a perfect fit for a 2^n dimensional target vector (assuming all functions in the feature space are independent)

TASK 3 – FRACTAL DIMENSION

Task 1.3 - Breakdown of Task

1. Apply an appropriate binarization procedure to create a binary image in which foreground pixels are set to 1 and background pixels to 0 (given)
2. Create scaling factors $S = \{1,2,\ldots,L-2\}$ where L is given by the width/height of the picture $w = h = 2^L$. Fit the ${2^{l+1}}^2$ boxes into the image and count in how many of those boxes there is a foreground pixel (a matrix entry with a 1). Do this for all $l \in S$ with the scaling $s_l = \frac{1}{2^l}$ and denote the count as n_l.
3. Now interpret the $(\log n_l)_l$ as target vector for the data points $\log_2 \frac{1}{s_l} = l$. Estimate the slope D of the equation $D \log \frac{1}{s_l} + b = \log n_l$

```python
def linregression(vecX, vecY):
    if vecX.shape==vecY.shape:
        # Phi is the feature matrix for the linear regression with the data vector x
        Phi = np.concatenate( ([np.ones(vecX.shape[0])], [vecX]), axis=0)
        return la.lstsq(Phi.T, vecY, rcond=-1)[0]
```

TASK 3 – FRACTAL DIMENSION

Task 1.3 - Application

Code:

```
def fractaldim(imgF):
    imgBin = binarize(imgF)
    L = round(np.log2(imgBin.shape[0]))
    # We store the L's in the scalings vector because the scaling of each iteration can be computed separately and then we don't need to compute the log
    scalings = np.array(range(2,L))
    counts = np.array(range(2,L))
    for exponent in scalings:
        # pow(2,L-exponent) is exactly the number of frames each with width pow(2,exponent) that can be fitted into the total width (same for height)
        count = 0
        for i in range(pow(2,L-exponent)):
            for j in range(pow(2,L-exponent)):
                # The condition of the if statement is only true if at least one pixel of the block in the i-th row and the j-th column is "True"
                if np.any(imgBin[ (i*pow(2,exponent)):((i+1)*pow(2,exponent)), (j*pow(2,exponent)):((j+1)*pow(2,exponent))]):
                    count += 1
        counts[exponent-2] = count
    scalings *= -1
    scalings += L
    return linregression(scalings, np.log2(counts))[1]
```

Output: Fractal dimension of the Tree: 1.8463900565472446

Fractal dimension of the Lightning: 1.4934991270542086