Principles of Machine Learning: Exercise 1

Alina Pollehn (3197257), Julian Litz (3362592), Manuel Hinz (3334548)
Felix Göhde (3336445), Felix Lehmann (3177181), Caspar Wiswesser (3221493)
Adrian Köring (3347785), Greta Günther (3326765), Linus Mallwitz (3327653)
Niklas Mueller-Goldingen (3363219), Jennifer Kroppen (???????)

06.11.2023

Task 2.1

- Given the following two rule sets, implement code that solves the least squares problem between X^T and y.
- Afterwards the code should calculate $\hat{y}$ using the calculated weights.

Task 2.1

Table: Rule 110

X^T			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	$+1$

Table: Rule 126

X^T			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	$+1$

Task 2.1

```python
# Solve the Least Squares Problem
w110 = la.lstsq(matXT, y110, rcond=None)

# Calculate yhat
yhat110 = matXT @ w110[0]
```

$$\hat{y}_{110} = \begin{pmatrix} +0.25 \\ -0.25 \\ -0.25 \\ -0.75 \\ +0.75 \\ +0.25 \\ +0.25 \\ -0.25 \end{pmatrix} \qquad \hat{y}_{126} = \begin{pmatrix} +1.57e-16 \\ -1.23e-32 \\ +1.57e-16 \\ -1.23e-32 \\ +1.23e-32 \\ -1.57e-16 \\ +1.23e-32 \\ -1.57e-16 \end{pmatrix}$$

Task 2.1

- The first calculation is numerically stable
- Even though the difference between the two rulesets is only one number, the second calculation becomes numerically unstable

Task 2.2

Implement a function phi which takes a vector x with n elements and realizes the following transformation: $\varphi : \{+1, -1\}^n \to \{+1, -1\}^{2^n}$

```
import itertools as it

def phi(x):
    n = len(x)
    # Generate all possible sets of x
    sets = it.chain.from_iterable(it.combinations(x, r) for r in range(n+1))
    # Multiply each set together to a single value and return these as an array
    return np.array([np.prod(s, dtype=int) for s in sets], dtype=int)
```

Task 2.3

Given the previously introduced function phi we compute the matrix $\phi^T = \begin{bmatrix} — & \varphi_0^T & — \\ — & \varphi_1^T & — \\ & \vdots & \\ — & \varphi_7^T & — \end{bmatrix}$

Solve the least squares problem between ϕ^T and y and calculate $\hat{y}$ using the calculated weights.

```
Phi = np.apply_along_axis(phi,1,x.T)

w110 = la.lstsq(Phi, y110 ,rcond=None)[0]
yhat110 = x.T @ w110
```

Task 2.3

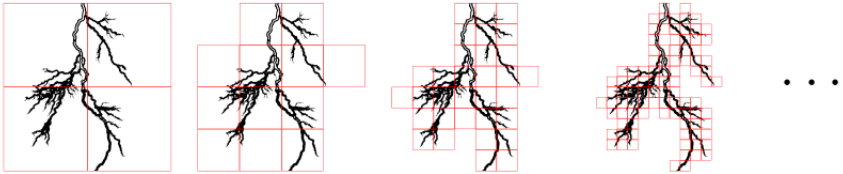
$$\hat{y}_{110} = \begin{pmatrix} +1 \\ -1 \\ -1 \\ -1 \\ +1 \\ -1 \\ -1 \\ +1 \end{pmatrix} \qquad \hat{y}_{126} = \begin{pmatrix} +1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ +1 \end{pmatrix}$$

We now observe that $\hat{y}_{110} = y_{110}$ and $\hat{y}_{126} = y_{126}$. We "trained" our "parameters" w with all possible inputs/"data" to archieve an output as close as possible to our rule so that we get a good output according to our rule for every input. We had to pay the price of computing a high dimensional feature map Phi. Because of this our "training" aka. the search for the least squares solution is highly complex (in contrast to the computing of the rule once the parameters have been found)

Fractal dimensions

1. Binarize the image
2. Partition the image into 2^l boxes for $l = 1, \ldots, L-2$,
3. Calculate the fractal dimension using linear regression

$$D \cdot \log\left(\frac{1}{s_l}\right) + b = \log(n_l)$$



$s_l = \frac{1}{2}, n_l = 4$ $s_l = \frac{1}{4}, n_l = 12$ $s_l = \frac{1}{8}, n_l = 32$ $s_l = \frac{1}{16}, n_l = 94$ . . .

Implementation: Box counting

```
def box_counting(img):
    w, h = img.shape
    n_ls = []
    # l runs from 1 to 9-2 =7
    for l in range(1, 8):
        n_l, s_l = 0, 1/2**l                            # setup counter and scale
        box_sizeW, box_sizeH = s_l * w, s_l * h         # get box sizes

        for box_w in range(0,(2**l)):                   # each l has 2**l boxes
            for box_h in range(0,(2**l)):
                #check if any value in the box is equal 1.
                #If so increment n_l by one
                if (np.any(img[int(box_w * box_sizeW): int((box_w+1) * box_sizeW),\
                               int(box_h * box_sizeH): int((box_h+1) * box_sizeH)]\
                            ==1)):
                    n_l+=1
        n_ls.append(n_l)
    return n_ls
```

Implementation: Calculating the fractal dimension

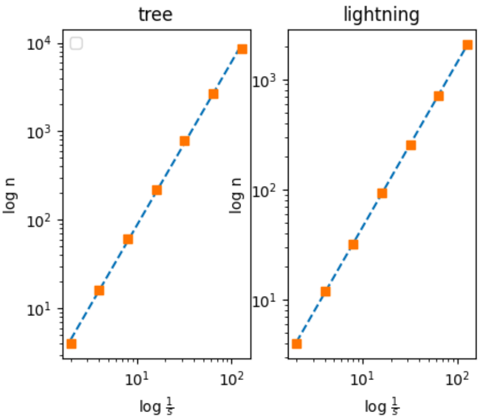
```
def slope(n_l):
    inverted_s_l=[2**l for l in range(1,8)]
    matX=np.vander(np.log(inverted_s_l),2,increasing=True)
    b,D=la.lstsq(matX, np.log(n_l),rcond=None)[0]
    return b,D
```

where np.vander generates the Vandermonde matrix

$$V = \begin{bmatrix} 1 & \log(2^1)^1 \\ 1 & \log(2^2)^1 \\ \vdots & \vdots \\ 1 & \log(2^{L-2})^1 \end{bmatrix}$$

Results

1. Fractal dimension
   - Tree: ≈ 1.846
   - Lighting: ≈ 1.493
2. The fractal dimension of the image "lighting.png" is higher



Learnings

1. Fractal dimensions
2. Fractal dimensions as a least squares problem using box counting
3. Application of least squares to a wider class of problems