MAVT Informatik II

vkirsanova
antakoulas

Version: 27. Juli 2023

Templates von Robin Frauenfelder - robinfr@ethz.ch.

Small Algos

Mittelwert berechnen

Function:

```
def mean(A):
  return sum(A) // len(A)
```

Runtime: $\mathcal{O}(n)$

Find smallest element

Function:

```
def smallest_elem(A):
  min A[0]
  for a in A[1:]:
    min = a if a < min else min
```

Runtime: $\mathcal{O}(n)$

Runtime Complexity Analysis

Big O Notation

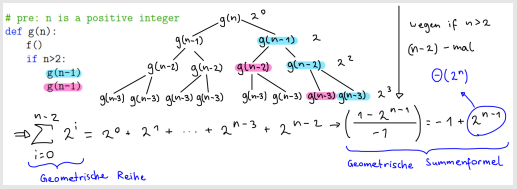
Sei g die Beziehung zwischen Zeit und Eingabegrösse für einen Algorithmus:

- if g doesn't grow faster than c*f:
  $g = \mathcal{O}(f)$
- if g grows as quickly as c*f:
  $g = \Theta(f)$
- if g doesn't grow faster than c*f:
  $g = \Theta(f)$

Useful Sum Formulae

- $\sum_{i=0}^{n-1} 1 = n = \Theta(n)$ $\quad\quad$ $\sum_{i=0}^{n} i = \frac{n\cdot(n+1)}{2} = \Theta(n^2)$
- $\sum_{i=0}^{n} i^2 = \frac{n\cdot(n+1)(2n+1)}{6} = \Theta(n^3)$
- $\sum_{i=0}^{n} i^a = \Theta(n^{a+1})$ $\quad\quad$ $\sum_{i=0}^{k} \frac{1}{2^i} = 2$
  wobei es egal ist was k ist wenn k ins unendliche geht.
- Geometrische Summenformel: $\sum_{k=0}^{\infty} q^k = (\sum_{k=0}^{n} q^k)\ n \in \mathbb{N} = (\frac{1-q^{n+1}}{1-q}) \approx \Theta(q(n))$

examples



Logarithm Rules

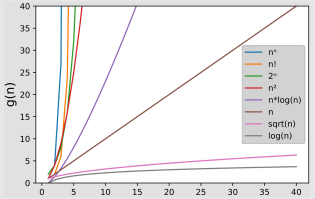
$a^x = N \text{ -> } log_a(N)$
$log_a(b) = \frac{log_c(b)}{log_c(a)}$

- $log_b(M\cdot N) = log_b M + log_b N$
- $log_b\left(\frac{M}{N}\right) = log_b M - log_b N$
- $log_b\left(M^k\right) = k\cdot log_b M$
- $log_b(1) = 0$
- $log_b(b) = 1$
- $log_b\left(b^k\right) = k$
- $b^{log_b(k)} = k$

Asymptotisches Wachstum von Funktionen

Funktionen in **zunehmender Reihenfolge** (von asymptotischen Wachstum):

1. $log(n)$
2. $\sqrt{n}$
3. n
4. $n \cdot log(n)$
5. $n^2\ (= \sum_{i=0} ni)$
6. 2^n
7. $n!$
8. n^n



$2^{log(n)} \approx n$

$2^{log_4(n)} = 2^{\frac{log_2(n)}{log_2(4)}} = (2^{log_2(n)})^{\frac{1}{log_2(4)}}$

Sort Algorithms

Number of Swaps & Comparisons

A: array
Worst-Case: Upper Bound
Best-Case: Lower Bound
Reverse Sorted: n, n-1, . . . , 1
Sorted: 1, 2, . . . , n

sort	# swaps			# comparisons			general
	case	A	runtime	case	A	runtime	
Bubble	worst	reverse sorted	Θ(n^2)	all	-	Θ(n^2)	Θ(n^2)
	average	-	Θ(n^2)				
	best	sorted	Θ(1) (0)				
Selection	worst	reverse sorted	Θ(n)	all	-	Θ(n^2)	Θ(n^2)
	average	-	Θ(n)				Θ(n^2)
	best	sorted	Θ(1) (0)				Θ(n)
Insertion	worst	reverse sorted	Θ(n^2)	worst	reverse sorted	Θ(n^2)	Θ(n^2)
	average	-	Θ(n^2)	average	-	-	Θ(n^2)
	best	sorted	Θ(1) (0)	best	sorted	Θ(n)	Θ(n)
Merge	all	-	Θ(nlog(n))	all	-	Θ(nlog(n))	Θ(n)
	Braucht zusätzlich Θ(n) Speicher für die Subarrays						
Quick	worst	*	Θ(nlog(n))	worst	*	Θ(n^2)	Θ(n^2)
	average	**	Θ(nlog(n))	average	**	Θ(nlog(n))	Θ(nlog(n))
	best	***	Θ(n)	best	***	Θ(nlog(n))	Θ(nlog(n))
Heap	all	-	Θ(nlog(n))	all	-	Θ(nlog(n))	Θ(nlog(n))

*pivot ends up being the min/max value each time
**pivot is chosen randomly
***pivot is always the median of the array

The general runtime of the code for each case is the sum of swaps and comparisons, the slower one will dominate.

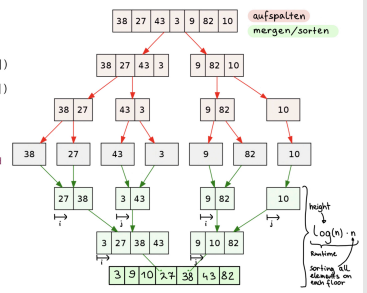
Divide and Conquer

"Divide and Conquer" ist ein Algorithmustyp, bei dem man das Problem rekursiv immer wieder in zwei oder mehrere gleich grosse Teilprobleme vom gleichen Typ teilt. Beispiele hierfür sind Mergesort und Quicksort.

Merge Sort

Ein Array sortieren, inde, man es in kleinere Subarray teilt (divide and conquer) und dann diese Subarrays sortieren.

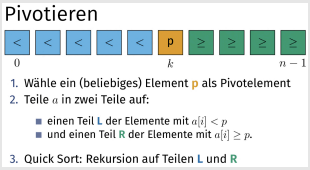
```
def merge_sort(a):
  if len(a) == 1:
    return a
  else:
    s1 = merge_sort
         (a[:len(a)//2])
    s2 = merge_sort
         (a[len(a)//2:])
    s = merge(s1,s2)
    return s

def merge(a1, a2):
  b, i, j = [], 0, 0
  while i < len(a1) and
        j < len(a2):
    if a1[i] < a2[j]:
      b.append(a1[i])
      i+=1
    else:
      b.append(a2[j])
      j+=1
  b+=a1[i:]
  b+=a2[j:]
  return b
```



Case	Description	Runtime
All cases	-	$\Theta(n \cdot log(n))$

Quick Sort

Ein Array sortieren, indem man das Array rekuriv immer wieder halbiert (divide and conquer) in zwei Teile: eine Hälfte, die Elemente grösser als den Pivot enthält und eine Hälfte, die Elemente A kleiner als den Pivot enthält.




```
                                  array   0      len(A)-1
def quicksort_median3_pivot(A, left, right):
    if left < right:
        # Median-of-three pivot selection
pivot selection system (if one is needed)
        mid = (left + right) // 2
        if A[mid] < A[left]:
            A[left], A[mid] = A[mid], A[left]
        if A[right] < A[left]:
            A[left], A[right] = A[right], A[left]
        if A[right] < A[mid]:
            A[mid], A[right] = A[right], A[mid]

        pivot = A[mid]

        i = left
        j = right-1

        while i < j:
            while i <= j and A[i] <= pivot:
                i += 1
            while i <= j and A[j] >= pivot:
                j -= 1
            if i < j:
                A[i], A[j] = A[j], A[i]

        A[left], A[j] = A[j], A[left]
        quicksort_median3_pivot(A, left, j)
        quicksort_median3_pivot(A, j+1, right)
```

Case	Description	Runtime
Worst-case	Pivot is the min/max value	$\Theta(n^2)$
Average-case	Pivot is chosen randomly.	$\Theta(n \cdot log(n))$
Best-case	Pivot is always the median of the array.	$\Theta(n \cdot log(n))$

Pivot is often the median of three elements:
Pivot = Median3(A[l],A[(l+r)//2],A[r])

Selection Sort

Ein Array sortieren, indem man jeweils das minimum von rechts aus wählt und swappt. Code:

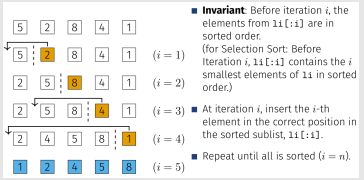
```
def selectionSort(A, l, r):
    for i in range(l,r):
        minJ = i
        for j in range(i+1,r):
            if(A[j] < A[minJ]):
                minJ = j
        if minJ != i:
            A[i], A[minJ] = A[minj], A[i]
```

Case	Description	Runtime
Worst-case	A is reverse sorted	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(n)$

Insertion Sort

```
def insertionSort(A, l, r):
    for i in range(l+1,r):
        j = i-1
        while j >= 0 and A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
            j -= 1
```

Case	Description	Runtime
Worst-case	A is reverse sorted	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(n)$



Binary Insertion Sort

```
# This function will return the index of element
# just greater than 'key' in Array from 0-N
def binarySearch(Array, N, key):
    L = 0
    R = N
    while(L < R):
        mid = (L + R)//2
        if (Array[mid] <= key):
            L = mid + 1
        else:
            R = mid
    return L

def binaryInsertionSort(Array):
    # We assume the 1st element of Array to be already sorted.
    # Now we start iterating from the 2nd element to the
    #last element.
    for i in range (1,len(Array)):
        key = Array[i]
        pos = binarySearch(Array, i, key)
        # 'pos' will now contain the index where 'key'
        #should be inserted.
        j = i
        #Shifting every element from 'pos' to 'i' towards right.
        while(j > pos):
            Array[j] = Array[j-1]
            j = j-1
        # Inserting 'key' in its correct position.
        Array[pos] = key
        print("The array after",i,"iterations =", *Array)
```

Case	Description	Runtime
Worst-case	A is reverse sorted	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(nlog(n))$

Bubble Sort

Code Expert:

```
def bubbleSort(A, l, r):
    for i in range(l,r):
        for j in range(r-i-1):
            if A[j] > A[j+1]:
                swap(A, j, j+1)
```

Case	Description	Runtime
All cases	-	$\Theta(n^2)$

Faster Version:

```
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n-1):
        # range(n) also work but outer loop will
        # repeat one time more than needed.
        # Last i elements are already in place
        swapped = False

        # optimize code, so if the array is already sorted, it
        # doesn't need to go through the entire process
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            # if we haven't needed to make a single swap,
            # we can just exit the main loop.
            return
```

Case	Description	Runtime
Worst-case	A is reverse sorted	$\Theta(n^2)$
Average-case	-	$\Theta(n)$
Best-case	A is already sorted	$\Theta(n)$

Heapsort

An array can be sorted by first being converted to a proper (min/max) heap and then that heap being converted to a sorted array.

```
def heap_sort(list):
  heapify(list) #turns list into valid heap

  for i in reversed(range(len(list))):
    list[0], list[i] = list[i], list[0]
    siftDown(list, 0, i)

  return
```

if I want a descending order: small ⟶ big
1) I make a minheap with heapify
2) I use minheap in siftdown

if I want an ascending order: big ⟶ small
1) I make a maxheap with heapify
2) I use maxheap in siftdown

Case	Description	Runtime
All	-	$\Theta(n \cdot log(n))$

Search Algos

Linear Search

Searching for the index of a specified element b in an **unsorted** Array a. Code:

```
def LinearSearch(a, b):
  for i, x in enumerate(a):
    if x == b:
      return i
  return "not_found"
```

Case	Description	Runtime
Worst-case	b is at the end of the array.	$\Theta(n)$
Average-case	finding successor.	$\Theta(n)$
Best-case	b is at the beginning.	$\Theta(1)$

Binary Search

Searching for the Index of a specified element b in a **sorted** array a. Code:

```
#a: array
#l: left index
#r: right index
#b: element to search for
def bin_search(a, l, r, b):
  if r < l:
    return None
  else:
    m = (l + r) // 2
    if a[m] == b:
      return m
    elif b < a[m]:
      return bin_search(a, l, m-1, b)
    else: #a[m]>b
      return bin_search(a, m+1, r, b)
```

Case	Description	Runtime
Worst-case	b is the max/min value.	$\Theta(log(n))$
Average-case	-	$\Theta(log(n))$
Best-case	b is exactly the median value.	$\Theta(1)$

Numpy

importieren

```
import numpy as np
```

liste zu numpy

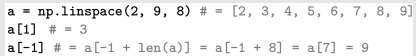
l = [1, 2, 3, 4]
a = np.array(l) # = [1, 2, 3, 4]
b = np.array(range(2,10,3)) # = [2, 5, 8]

Länge und Grösse

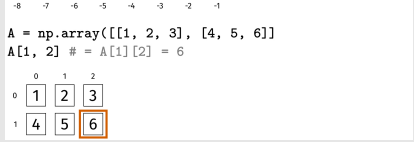
A = np.array([[1, 2, 3], [4, 5, 6]])
len(A) # = 2 (Anzahl Zeilen
len(A) # = 3 (Anzahl Spalten)
A.size # = 6 (Anzahl Elemente)

Zugriff auf Element

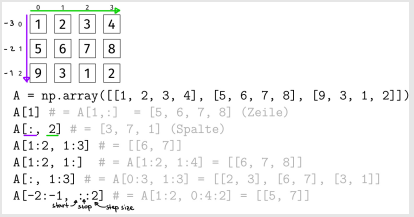
```
a = np.linspace(2, 9, 8) # = [2, 3, 4, 5, 6, 7, 8, 9]
a[1] # = 3
a[-1] # = a[-1 + len(a)] = a[-1 + 8] = a[7] = 9
```




```
A = np.array([[1, 2, 3], [4, 5, 6]]
A[1, 2] # = A[1][2] = 6
```



Slicing Numpy Matrices



Zufallszahlen

R = np.random.random(10) # zehn Zufallszahlen aus [0,1)
R = np.random.uniform(-1, 1, 5) # fünf Zufallszahlen aus [-1, 1) (bis und ohne 1)
R = np.random.randint(1, 7, 10) #zehn Würfelwürfe (ganze Zahlen)

Printing

print(np.array([[1, 2], [3, 4], [5, 6]]))
# [[1 2]
# [3 4]
# [5 6]]

linspace

a = np.linspace(-4, -2, 3) #[-4, -3, -2]
(start (default 0), stop (bis und mit), num of steps (this -1))
Schrittgrösse: $\frac{stop-start}{num-1}$

arange

b = np.arange(-2, 6, 2) #[-2, 0, 2, 4]
np.arange(start, stop, step)
np.arange(start, stop) # step = 1
np.arange(stop) # start = 0, step = 1
c = np.arange(5) #[0, 1, 2, 3, 4]

array

A = np.array([[1, 2, 3]]*2) #[[1, 2, 3], [1, 2, 3]]

filter

g = b + c > 3
b[g] #[4, 7]

Elementenweise Operationen

A = np.array([[2, 3, 4], [6, 7, 6]])
B = np.array([[1, 9, 1], [2, 3, 9]])
• **Addition**:
A + 1 #[[3, 4, 5], [7, 8, 7]]
A + B #[[3, 12, 5], [8, 10, 15]]
• **Skalarmultiplikation**:
A * 2 #[[4, 6, 8], [12, 14, 12]]
A * B #[[2, 27, 4], [12, 21, 54]]
• **Potentz**:
A**4 #[[16, 81, 256], [1296, 2401, 1296]]
• **Matrixmultiplikation** (Dimensionen müssen stimmen):
A @ np.array([[1, 4], [3, 4], [4, 6]]) #[[27, 44], [51, 88]]
A.dot([[1, 4], [3, 4], [4, 6]]) #[[27, 44], [51, 88]]
• **Skalarprodukt**:
a = [0, 1, 2] b = [0, 5, 20] scalar_prod = (a * b).sum() # = [0, 5, 20].sum() = 25
a.dot(b) # = 25

min, max, sum, mean, std

a = np.array([1, 2, 3, 4, 5]) A = np.array([[2, 3, 4], [6, 7, 6]])
a.min() # = 1
a.max() # = 5
a.sum() # = 15
np.sum(A, axis = 0) # summiert spalten = [8, 10, 10]
np.sum(A, axis = 1) # summiert zeilen = [9, 19]
np.mean(a) # = a.sum()/a.size = 1
np.std(a) # standard deviation = np.sqrt(np.mean((a - np.mean(a))^2))

Pandas

importieren

```
import pandas as pd
```

CSV Datei einlesen mit Pandas

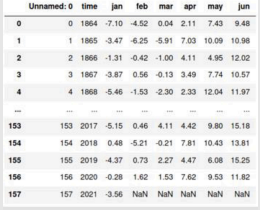
(Datei im gleichen Ordner gespeichert wie das Program)
climate = pd.read_csv("climate.csv", sep=",", index_col=0, usecols=["time", ...])

- sep: what elements in the csv file are separated by.
- index_col: what the index column will be.
- usecols: what columns of the csv data will be selected.

Die Index Spalte verändern

Die Index Spalte ändern (erstellen einer Kopie) climate2 = climate.set_index("time")

Pandas Dataframe:



	Unnamed: 0	time	jan	feb	mar	apr	may	jun
0	0	1864	-7.10	-4.52	0.04	2.11	7.43	9.48
1	1	1865	-3.47	-6.25	-5.91	7.03	10.09	10.98
2	2	1866	-1.31	-0.42	-1.00	4.11	4.95	12.02
3	3	1867	-3.87	0.56	-0.13	3.49	7.74	10.57
4	4	1868	-5.46	-1.53	-2.30	2.33	12.04	11.97
...	...	...	...	...	...	...	...	...
153	153	2017	-5.15	0.46	4.11	4.42	9.80	15.18
154	154	2018	0.48	-5.21	-0.21	7.81	10.43	13.81
155	155	2019	-4.37	0.73	2.27	4.47	6.08	15.25
156	156	2020	-0.28	1.62	1.53	7.62	9.53	11.82
157	157	2021	-3.56	NaN	NaN	NaN	NaN	NaN

climate

time	Unnamed: 0	jan	feb	mar	apr	may	jun
1864	0	-7.10	-4.52	0.04	2.11	7.43	9.48
1865	1	-3.47	-6.25	-5.91	7.03	10.09	10.98
1866	2	-1.31	-0.42	-1.00	4.11	4.95	12.02
1867	3	-3.87	0.56	-0.13	3.49	7.74	10.57
1868	4	-5.46	-1.53	-2.30	2.33	12.04	11.97
...	...	...	...	...	...	...	...
2017	153	-5.15	0.46	4.11	4.42	9.80	15.18
2018	154	0.48	-5.21	-0.21	7.81	10.43	13.81
2019	155	-4.37	0.73	2.27	4.47	6.08	15.25
2020	156	-0.28	1.62	1.53	7.62	9.53	11.82
2021	157	-3.56	NaN	NaN	NaN	NaN	NaN

climate2

Spalten umbenennen

• **Mithilfe der "rename" Funktion**:
climate = climate.rename(columns="time":"date", ...)
# "old_index_name":"new_index_name"
• **Die Spaltennamen direkt setzen**:
data.columns = ["Date", "January", "February",...]
# needs to be the same length as the number of columns

Auf Dataframe Elemente zugreifen

• **Auf eine einzelne Spalte zugreifen**:
climate["feb"] # gets the column "feb"
• **Auf mehrere Spalten zugreifen**:
climate[["jan", "mar"]] # gets the columns "jan", "mar"
• **Auf eine einzelne Spalte mit Index zugreifen**:
climate.iloc[3] # gets row 3
• **Verschiedene Zeilen**:
climate.iloc[1:4] # gets rows 1 to 3
• **Auf eine Subtabelle zugreifen mit Indizes**:
climate.iloc[4:7,1:2] # gets rows 4,7 with data only from column 1
• **Auf eine Subtabelle zugreifen mit Spaltenindexwerten und**

Spaltennahmen:
climate2 = climate_index("time")
climate2.loc[1864:1868,"jan":"mar"]
includes the rows labeled with 1866 until and including 1868, the columns from "jan" until and including "mar"
Auf ein einzelnes Element zugreifen:
climate["jan"][3] # gets the element in column "jan" in row 3

Dataframes filtern

- **Zeilen filtern**:

climate[climate["jan"]>2]
filters out the rows with values in the "jan" column less that 2
- **Beispiel: Alle Einträge in "jan" und Werten über 2**:

climate["jan"][climate["jan"]>2]

Mit Invalid Data umgehen

- **Alle Werte in einer Spalte zu numeric konvertieren**:

data[column] = pd.to_numeric(data[column], errors = "coerce")
# converts all the values to numeric values.
errors = "coerce" -¿ converts values which cannot be converted to NaN.
- **Alle Zeilen mit NaN Werten löschen**:

data.dropna(axis = 0, how = äny")
# how = "any" -> delete row if any value is NaN.
how = "all" -> delete row if all values are NaN
axis = 1 -> delete column instead of row
- **Alle Einräge mit NaN Werten mit anderem Wert ersetzen**:

data.fillna(0) #fill any NaN entries with 0

Dataframes verändern

- **Spalte hinzufügen**:

climate["new_col"] = climate["time"] + climate["jan"]
"new_col" is a new column who's values are those of the "time" and "jan" column added
- **Spalte löschen**:

climate = climate.drop(columns=["time"])
# delete the "time" column
- **Zeile hinzufügen**:

d = "mar":34, "jan:23
climate.append(d, ignore_index = True)
#adds another for with the values 34 for "mar and 23 for "jan". Other entries are NaN
- **Zeile löschen**:

climate.drop(climate.index[0]) # deletes row 0
- **Dataframe transponieren** (Zeilen und Spalten vertauschen):

climate = climate.T

Daten in Dataframes analysieren

- **Alle einträge in jeder Spalte aufaddieren**:

climate.sum()
- **Das Maximum aller Einträge in jeder Spalte**:

climate.max()
- **Ein Dataframe erstellen, welches das Maximum und Summe für jede Spalte zusammenfasst**:

climate.agg(["max","sum"])
#A dataframe containing the same columns as climate with row 0 containing the max of the column and row 1 containing the sum of the column. The strings in the list shoukd be names of valid pandas Serier functions.
- **Statistische Informationen über jede Spalte bekommen**:

climate.descrime() #includes a variety of statistical measures
- **Ein Dataframe sortieren anhand von Einträgen in spezifischen Spalte(n)**:

climate = climate.sort_values(["time", "jan"], ascending = False) #sorts the rows by "time" in descending (from highest to lowest) order. If two entries for "time" are equal, then the rows are sorted by "jan"
- **Ein Dataframe aufteilen in Gruppen anhand von spezifischen Spalten und mathematische Operationen auf jede Gruppe anwenden**:

data.groupby("column").sum()
# groups data based on the entries for "column" and calculates the sum for each group.
data.groupby("column").max()
#groups data based on the entries for "column" and calculates the max for each group.

Matplotlib

Matplotlib ist eine Python Package, mit der man viele Sachen visualisieren kann (Funktionen, Daten, Animationen).

matplotlib importieren

```
import matplotlib.pyplot as plt
```

Line Plots

- **Zwei Numpy Arrays grafisch darstellen, X -> Wert x-Achse, Y -> Wert y-Achse**:

X = np.linspace(0,2*np.pi,100)
Y = np.sin(X)

fig, ax = plt.subplots()
ax.plot(X,Y)

Scatter Plots

- **Zwei Numpy Arrays grafisch darstellen, X -> Wert x-Achse**:

X = np.arange(1,9)
Y = np.array([1,2,31,2,3,1,2])

fig, ax = plt.subplots()
ax.scatter(X,Y)

Histogramm Plots

- **Um ein Histogramm zu erstellen**:

fig, ax = plt.subplots()
X = np.random.randint(0, 100, 500)
#low: 0, high: 100, size: 500
ax.hist(X, bins=10)
plt.show()

Graph Styling

fig, ax = plt.subplots()
- **Um einen Titel hinzuzufügen**:

ax.set_title("title")
- **x-Achse beschriften**:

ax.set_xlabel("x label name")
- **y-Achse beschriften**:

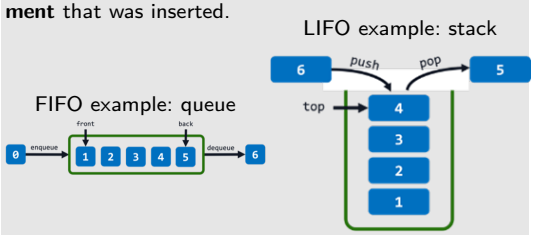
ax.set_ylabel("y label name")
- **Eine legende hinzufügen**:

ax.legend()
#requires that you labeled your plots, i.e.: when calling ax.plot(X,Y, label="name of function").

FIFO_LIFO

Some data structures can be categorized as FIFO (first in, first out) or LIFO (last in, first out).

FIFO data structures offer **fast access to the first element** that was inserted.
LIFO data structures offer **fast access to the last element** that was inserted.



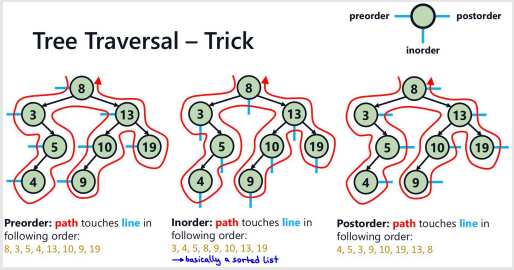
Trees Generally

Traversal Rules

- Given an inorder traversal: there is no possible representation of the tree if the sequence is not in ascending order.**Representation is not unique.**
Example: 1 2 4 3 has no representation as a BST.
- Given a preorder traversal: there is no possible representation of the tree if there is not a way to (recursively for the new subsequences as well) place the first number in the sequence so that the numbers to the left are smaller and that the numbers to the right are greater. **Representation is unique.**
Example: 4 3 1 2 8 6 5 7
3 1 2 4 8 6 5 7
1 2 3, 6 5 7 8
1 2, 5 6 7 − > valid
- Given a postorder traversal: there is no possible representation of the tree if there is not a way to (recursively for the new subsequences as well) place the last number in the sequence so that the numbers to the left are smaller and the numbers to the right are greater. **Representation is unique.**
Example: 1 3 2 5 6 8 7 4
1 3 2 4 5 6 8 7
1 2 3, 5 6 7 8
5 6 − > valid

Tree Traversal Tricks

Traversing a tree: Visiting every element of the list once



Pre-/Postorder: check if you can place the First/Last element into the list so that all on the left are smaller and all on the right are bigger. If not possible no tree possible. Pre-/Postorder lists allow only one unique tree.
Inorder: Allow an infinite number of combinations (Ex: degenerate trees)

Ordering Tree Lists

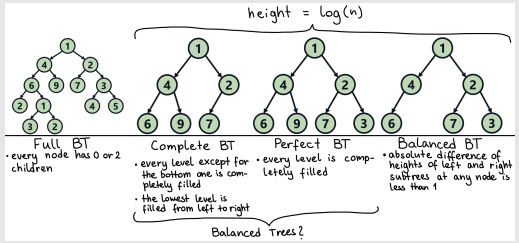
Now that we have a fully defined tree with working class functions we can do things with it. A common demand is to gather the keys of the list in preorder, inorder or postorder:

```
def preorder_list(node: Optional[Node]) -> list:
  if node is None:
    return []
  return [node.key] + preorder_list(node.left) +
  preorder_list(node.right)

def postorder_list(node: Optional[Node]) -> list:
  if node is None:
    return []
  return [node.key] + postorder_list(node.right) +
  postorder_list(node.left)

def inorder_list(node: Optional[Node]) -> list:
   if node is None:
    return []
  postorder_list(node.left) + return [node.key] +
  postorder_list(node.right)
```

Binary Tree

Binary Tree: Every node has at most two children

Types of Binary Trees



Binary Search Trees (BSTs)

Class Node and Class Tree

In a BST the binary rules apply + all the elements to the right of every parent must be bigger than it and all to the left must be smaller.
There are two classes that define a BST. The class Node and the class Tree. The class Node is a container which may contain the following and more:

```
class Node:
  def __init__(
    self, key: int, left: Optional["Node"] = None,
    right: Optional["Node"] = None
  ):
    self.key = key
    self.left = left
    self.right = right
```

In the Tree class we define the root attribute of the Tree (a pointer to the parent of the parents) which is implemented on the root node of the tree. And other functions that help with creating and modifying a BST.

Searching for Node (Key) (into a BST)

Then there is the search Function that searches for a key:

```
def search(self, key: int) -> Optional[Node]:
  """Search for the node with the specified key in the
  tree and retur it, or None if there is no such node."""
  n = self.root
  while n is not None and n.key is not key:
    if key < n.key:
      n = n.left
    else:
      n = n.right
  return n
```

Case	Description	Runtime
Worst-case	tree (BST) is degenerated	$\Theta(n)$
Average-case	tree (BST) is balanced	$\Theta(log(n))$
Best-case	Key is the root of the tree	$\Theta(1)$

Adding Node (Key)

When creating a binary search tree we insert elements one by one. If the key of the element to be inserted is bigger than the key that it is on we move one to the right and so on.
Code Implementation:

```
def add(self, key: int) -> bool:
  """Add a node with the specified key to the tree
  if it does not already exist in the tree. Return
  True if the node was added, and False otherwise."""
  if self.root is None:
    self.root = Node(key)
    return True
  n = self.root
  while n.key is not key:
    if key < n.key:
      if n.left is None:
        n.left = Node(key)
        return True
      n = n.left
    else:
      if n.right is None:
        n.right = Node(key)
        return True
      n = n.right

  return False
```

typically the while condition is adapted to the specific requirements, the rest can remain as is.

Case	Description	Runtime
Worst-case	tree (BST) is degenerated	$\Theta(n)$
Average-case	tree (BST) is balanced	$\Theta(log(n))$
Best-case	tree is empty	$\Theta(1)$

Removing Node - Finding Successor

A function that removes a node with specified key.

1. If the node has no children set Node = None
2. If node has only one child (/one subtree) replace the node with that child
3. If Node has two children, find its symmetric successor (the next largest node).

What is important here is finding the Successor for the deleted node and properly attaching it.
The successor to replace the deleted node will be:

- it will either be the leftest thing on the right subtree (the smallest bigger key existing)
- or the rightest on the left subtree (the biggest smaller key existing)

We use the first method.

Code:

```
def deleteNode(root,key):
  if root is None:
    return None
  #Find the node to be deleted
  if key < root.val:
    root.left = deleteNode(root.left, key)
  elif key > root.val:
    root.right = deleteNode(root.right, key)
  else:
    #Case 1: Node has no children
    if root.left is None and root.right is None:
      root = None
    #Case 2: Node has no children
    elif root.left is None:
      root = root.right
    elif root.right is None:
      root = root.left
    #Case 3: Node has two children
    else:
      successor = findSuccessor(root)
      root.val = successor.val
  return root

def findSuccessor(start_node):
  parent = start_node
  node = start_node.right
  while node.left is not None:
    parent = node
    node = node.left
  parent.left = node.right
  return node
```

Case	Description	Runtime
All	Runtime is dominated by finding successor.	$O(n)$ h is the height of the tree

count follow up nodes of a node (its total offspring)

Just add the attribute family to the class node and then add a line with n.family += 1 right after the while condition of the add function.

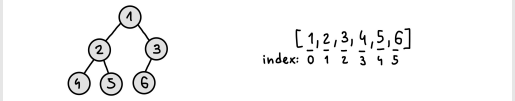
```
class Node:
  def __init__(self, reservation_time: int):
    self.key = reservation_time
    self.left = None
    self.right = None
    self.family = 1
```

Heaps

Binary Heaps

min heap: Parent smaller than children
max heap: Parent bigger than children

implemented as a list structure with all the rows of the heap one after the other:



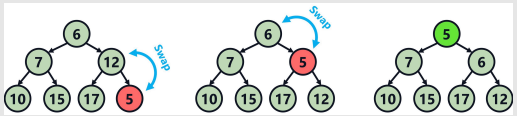
for easier access in code:

- parent = (i-1)//2
- left child = 2i + 1
- right child = 2i + 2
- height of heap = $log_2(N+1)$ with N the len(list)

where i is the list index

SiftUp

When creating a heap (convert list to heap list) we insert elements from left to right. Whenever a new element violates the heap rules it is swapped to the top until it complies with the rules.




```
def heapify(list):                          > minheap
  for i in range(len(list)):                < maxheap
    while list[(i-1)//2] < list[i] and i > 0:
      swap(list, (i-1)//2, i)
      i = (i-1)//2
    return
```

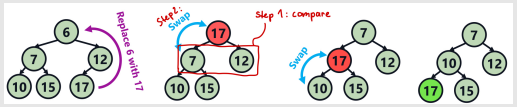
or

```
def heapify(list):
  n = len(list_a)
    for i in range(n//2 - 1,-1,-1):
      shift_down(list_a, i, n)
    return
```

SiftDown (deleting top element)

When inserting an element from the bottom right of heap (end of the list) to the very top of the heap with the root we then need to reorder the heap correctly by pushing that inserted element down where it fits according to heap rules.

- SiftDown for max-heap: always swap in direction of greater child
- SiftDown for min-heap: always swap in direction of smaller child




```
def SiftDown(a, i, m):                  maxheap
  while 2*i + 1 < m:                    other way around
    j = 2*i + 1                         for minheap
    if j + 1 < m and a[j]<a[j + 1]:
      j = j + 1
    if a[i] >= a[j]:
      break
    a[i], a[j] = a[j], a[i]
    i = j
```

a: list
i: index
m: size → len(list), with the element to be replaced already having been deleted from the list.

Sort Heap List

It is easier to sort a list in ascending or descending order after it has been turned into a heap. 'a' is a heap list.

```
def SortHeap(a):
  heapify(a)
  for i in reversed(range(len(a))):
    swap(a[0],a[i])
    SiftDown(a, 0, i)
  return
```

Removing and Inserting Heap Elements

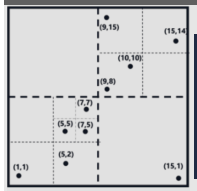
Inserting: Visually an element is inserted at the very bottom right.
Deleting: When an element is removed from the heap it is swapped with the last element.

```
def removeMax(heap):
  if len(heap) == 0:
      return None
  max_val = heap[0]
  heap[0] = heap[-1]
  heap.pop()
  SiftDown(heap, 0, len(heap)-1)
  return max_val
```

Binary Heap Complexities

- Worst Case building complexity:$\Theta(nlog_2(n))$
- Getting max or min element: $\mathcal{O}(1)$
- Inserting element into Max- or Minheap: $\mathcal{O}(log_2 N)$
- Removing maximum or minimum Element: $\mathcal{O}(log_2 N)$

Quad Trees



QuadTree mit max_cap = 2

```
class QuadTree:
    def __init__(self, l, u, max_cap):
        self.l = l #quadrant coordinate
        self.u = u #quadrant coordinate
        self.m = max_cap #max capacity
        self.points = [] #points in quadrant
        self.children = None #subquadrants
        self.count = 0 #point count
```

Beispiel einer Quadtree Implementation:

```
class QuadTree:
  def __init__(self, l, u, max_cap)
    self.l = l #coordinate of lower left corner
    self.u = u #coordinate of upper right corner
    self.m = max_cap
    self.points = []
    self.children = None
    self.count = 0 #total number of points within this
                   #quadtree

  #we insert points into the quadtree until the quadtree's
  #capacity is exceeded, in which case we add 4 children
  #to the quadtree and put the points in respective
  #correct child
    def insert(self, point):
        if self.children is not None:
            index = self.get_index(point)
            if index is not None:
                self.children[index].insert(point)
                self.count += 1
                return
        self.points.append(point)
        if len(self.points) > self.m:
            self.subdivide()
            for point in self.points:
                index = self.get_index(point)
                if index is not None:
                    self.children[index].insert(point)
                    self.count += 1
            self.points = []
```

Hash Tables

General Layout

```
def our_hash(E):
  def our_hash(word):
  """Return the hash value of the string word for our
  custom hash function.
  Args:
    word <str>: a word we want to turn into an int
  Return:
    <int>: hash value of word
  """
  return ord(word[0])

def create_hash_table(l, M):
  """Return the hash table of size M resulting from
     inserting elements from l in that order.
  Args:
    l : list elements to be inserted into hash table
    M <int>: size of hash table
  Return:
    <list[list[str]]>: hash table with chaining
  """
  return [[]]
```

Hash-Tabellen sind eine Datenstruktur, die schnellen Zugriff auf Elemente ermöglicht.

- Unsortierte, ungeordnete Daten.
- Schnelle Suche.

Die Idee hinter der Implementierung besteht darin, eine "Hash-Funktion" zu verwenden, um einen Index/Adresse aus dem Element zu erhalten und das Element dort zu speichern.

Operation	Best Case Complexity	Worst Case Complexity
Search	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Removal	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Hash-Table With Chaining

if two elements have the same value determined by our_hash the current element is just chained to the elements already there.
Eine Linked List an jedem Eintrag.

```
def our_hash(word):
  return ord(word[0])

def create_hash_table(l, M):
  hash_table = [ [] for x in range(M)]

  for x in l:
    index = our_hash(x) % M
    if x not in hash_table[index]:
      hash_table[index].append(x)

  return hash_table
```

Hash-Table With Probing

Der nächste verfügbare Index wird gewählt.

```
def create_hash_table(l, M):
  M_list = [[] for _ in range(M)]

  for x in l:
    current = our_hash_function(x)%M
    counter = 0
    for y in range(M):
      if len(M_list[current]) == 0:
        M_list[current].append(x)
        break
      if M_list[current][0] == x:
        break
      current = (current + 1)%M

  return [item[0] if item else None for item in M_list]
```

Classes

Overloading Operators and Functions

Overloading Operators:

Operator	Meaning	Class Method
+,+=	Addition	__add__, __iadd__
-	Subtraction	__sub__
*	Multiplication	__mul__
/	Division	__truediv__
//	Ganzzahldivision	__floordiv__
%	Modulo (Rest)	__mod__
**	Exponentiation	__pow__

Operator	Meaning	Class Method
<	less than	__lt__
<=	less than or equal	__le__
>	greater than	__gt__
>=	greater than or equal	__ge__
==	equal to	__eq__
!=	not qual to	__ne__

Overloading Functions:

Function	Meaning	Class Method
print()	printing as str	__str__

Inheritance

Superclass: a Class that is defined and then input as a default member in another Class (Subclass).
We use super().__init__() to initialize the Superclass as an object inside of a Subclass. Like so:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def honk(self):
        return f"{self.model} is honking!"
```

Superclass inside of Subclass:

```
class ElectricCar(Car):
    def __init__(self, brand, model):
        super().__init__(brand, model) #calls __init__ of superclass Car

    def charge(self):
        return f"{self.model} battery is charging."
```

Implementing In-Class Functions

ATTENTION:
When overwriting things like addition with self and other you SHOULD return the outcome as a Type Class instance itself. Otherwise it is an undefined entity and you can't really implement things like personalized print() on it.

```
class CustomString:
    def __init__(self, string):
        self.string = string

    def __str__(self):
        return self.string

    def __add__(self, other):
        result_string = ""
        for i in range(len(self.string)):
            if i < len(other.string):
                result_string += self.string[i] + other.string[i]
            else:
                result_string += self.string[i]
        return CustomString(result_string)

s1 = CustomString("abcd")
s2 = CustomString("123")
print(s1 + s2)
```

DP

Fibbonaci (DP + Memo)

Using a List Bottom-up:

```
def fib_DP(n): #n the "number" I want?
  # create table
  F = [None] * (n+1)
  # border cases
  F[0] = 1
  F[1] = 1

  #Bottom-Up for loop
  for i in range(2, n+1):
    F[i] = F[i-1] + F[i-2]

  # return last value
  return F[n]
```

Using recursion with memorization (into a list) Top-down:

```
def fibonacci(n, memo = None):
#memo is a list for memorization
  if memo is None:
    memo = [1,1] + [None]*(n-1)
  if memo[n] is not None:
    return memo[n]
  else:
    memo[n-1] = fibonacci(n-1, memo)
    memo[n-2] = ibonacci(n-2, memo)
    memo[n] = memo[n-1] + memo[n-2]
    return memo[n]
```

Golomb Numbers (Memo)

$$\text{ReLU(x)} = \begin{cases} 1 & n = 1 \\ 1 + S(n - S(S(n-1))) & n > 1. \end{cases}$$

just recursion:

```
def golomb(n):
  """
    pre: integer n > 0
    post: return the nth Golomb number
  """
  if n == 1:
    return 1
  else:
    return 1 + golomb(n-golomb(golomb(n-1)))
```

memorization:

```
def golomb(n, memo = None):
  if memo is None:
    memo = [1] + [None]*(n-1)
  if memo[n-1] != None:
    return memo[n-1]
  else:
    memo[n-1] = 1 + golomb(n-golomb(golomb(n-1, memo),
    memo), memo)
    return memo[n-1]
```

Aufgabenplanung

```
def tasks(w, t): #two lists of equal lengths
  n = len(w)
  s = [None] * (n+1)

  for i in range(n, -1, -1):
    if i == n:
      s[i] = 0
    else:
      finish_time = min(n, i + t[i]) #because there's no point
      #considering anything further than n (the end of list s)
      s[i] = max(w[i] + s[finish_time], s[i+1])
  return s[0]
```

Stab Schneiden, Stückgrössen

```
#n: L nge des stabes
#w: Liste mit Werten f r Stabst ckchenl nge
def cuts(n, w):
  s = [None]*(n+1)
  L = [None]*(n+1)

  for i in range(n+1):
    if i==0:
      s[i]=0
    else:
      option = 1
      value = w[1] + s[i-1]
      for j in range(1, i+1):
        new_option = j
        new_value > value:
          option = j
          value = new_value
    s[i]=value
    L[i]=option

  i=n
  cut=[]
  while i >= 1:
    cut.apend(L[i])
    i -= L[i]
  return (s[n], cut)
```

Karotten in nxn Matrix, bester Weg

```
def longest_path_dp(grid):
  n, m = len(grid), len(grid[0])
  s = [[None] * m for _ in range(n)]
  decision = [[None] * m for _ in range(n)]

  #fill out the DP table:
  for i in range(n-1, -1, -1):
    for j in range(m-1, -1, -1):
      if i == n - 1 and j == m - 1:
        s[i][j] = grid[i][j] decision[i][j] = "Stop"
      elif i == n - 1 and j < m-1:
        s[i][j] = grid[i][j] + s[i][j+1] decision[i][j] = "East"
      elif i < n - 1 and j == m - 1:
        s[i][j] = grid[i][j] + s[i+1][j] decision[i][j] = "South"
      else:
        s[i][j] = max(grid[i][j] + s[i][j+1],
                      grid[i][j] + s[i+1][j])
      if s[i][j] == grid[i][j] + s[i][j + 1]:
        decision[i][j] = "East"
      else:
        decision[i][j] = "South"

  #reconstruct that best path:
  path, i, j = [], 0, 0
  while (i, j) != (n-1, m-1):
    path.append(decision[i][j])
    if decision[i][j] == "East":
      j=j+1
    else:
      i=i+1

  return s[0][0], path
```

Can it be built?

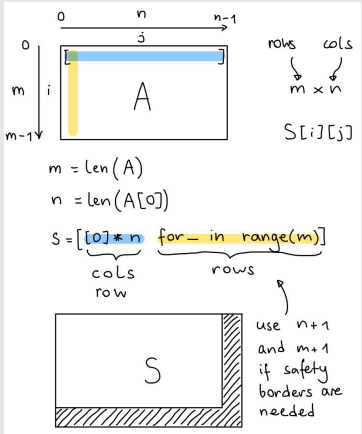
can we build an n length stick out of the provided [blocks]

```python
def what_is_possible(n, blocks):
  s = [True] + [False] * (n)

  for i in range(1,n+1):
    for b in blocks:
      if i-b < 0: #it passed without this too
        continue #stop this iteration of the loop and go
                 #onto the next one
      if s[i-b]:
        s[i] = True
        break

  return s
```

King's Way

```python
def stonesearch(A): #A is the field we are provided with
        m = len(A)
        n = len(A[0])
        S = [[0]*(n+1) for _ in range(m+1)]
        directions = [[None]*n for _ in range(m)]
        #j: cols, i: rows

        for j in range(n-1,-1,-1):
            for i in range(m-1,-1,-1):
                A[i][j]+max(S[i][j+1], S[i+1][j])
                if S[i][j+1] > S[i+1][j]:
                    directions[i][j] = "right"
                else:
                    directions[i][j] = "down"

        path = []
        i, j = 0, 0
        while directions[j][i] not None:
            path.append(directions[j][i])
            if directions[j][i] == "right":
                j += 1
            else:
                i += 1

        return S[0][0], path
```

Working with/making 2D-lists



Längste gemeinsame Teilfolge (Memorization)

Seq1: INbFORMaATInK
Seq2: I3NFOkRMATzIK
LCS: 10 (Längste gemeinsame Teilfolge INFORMATIK hat Länge 10)

```python
def S(seq1, i, seq2, j, A):
  if i == len(seq1) or j == len(seq2):
    return 0
  if A[i][j] != 0:
    pass
  elif seq1[i] == seq2[j]:
    A[i][j] = 1 + S(seq1, i + 1, seq2, j + 1, A)
  else:
    A[i][j] = max(S(seq1, i + 1, seq2, j, A), S(seq1, i, seq2,
                  j + 1, A))
  return A[i][j]

def lcs(seq1, seq2):
  A=[[0]*(len(seq2)) for _ in range(len(seq1))]
  return S(seq1, 0, seq2, 0, A)
```

ML

ML Code

A Model can be trained and validated in its accuracy by using given data:

axis = 0: is the default (doesn't need to be written). Means that the row at the provided index or label will be dropped.

axis = 1: Feature of drop. Means that the column at the provided index or label will be dropped.

```python
import pandas as pd
import numpy as np #do I need this???????

#read in data and separate into X and y
df = pd.read_csv("data.csv") #df: data frame
X = df.drop(["target"], axis=1) #all(cols) except for target
y = df["target"] #only the col target

#Split  data into training and testing part
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)
#test_size says for many % of df I want to leave to test
#with in the end

#choose and train your model
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier() #select model
model.fit(X_train,y_train) #train model

#Validate the accuracy of your CLASSIFICATION model by
#predicting on test set
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)
#chech what y my now trained model
#will predict for the X_test set
validation_score = accuracy_score(y_test, y_pred)
#validate model: see how much the predicted y of the test
#data deviates from the actual y of the test data

#Validate the accuracy of your REGRESSION model
from sklearn.metrics import r2_score
y_pred = model.predict(X_test)
validation_score = r2_score(y_test, y_pred)
```

Hidden Test Set

Sometimes they want us to see how accurate our model is using a hidden test set where we do not know the y_hidden but we can see how accurate our model performs on guessing y_hidden for the respective X_hidden.

```python
X_final = pd.read_csv("X_final.csv")
y_final = model1.predict(X_final) #model.predict returns
#a data frame because X_final is a data frame?
#why do I need to put my prediction into a data frame here?
return y_final
```

Helping the ML Model

```python
#following example is the spirals combination:
from sklearn.neural_network import MLPClassifier

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.1, random_state = 31)

model = MLPClassifier(hidden_layer_sizes=
(8, 6, 7, 6, 7, 5), max_iter=5000, random_state = 85)
model.fit(X_train, y_train)
```

max_iter
All ML models have an inbuilt loss function and a default number of iterations (200). During fitting the model will loop over the test set and if by the end of that the loss function won't be satisfied (training loss: 0.001) it will end in an error. If that happens we can increase the number of loops/iterations and hope that the model will manage to satisfy the loss function this time. PS: 1.000 Epoch = 1000 iter.

- works for: LogisticRegression, MLPClassifier, other?

hidden_layer_sizes
Just as before (except this time specifically for MLP) a neural network Model has a set number of neuron layers, namely one default layer hidden_layer_sizes = (100,). But sometimes that isn't enough/efficient, so we can increase the amount of those layers and tweak the number of neurons in them in order to satisfy the R2-Score requirements.

- works for: MLPClassifier, other?

random_state
The train_test_split shuffles the data points randomly every time the function is called. Sometimes the random state chosen for training will not be as efficient as another random state. This is why we can just choose a random state that works better and set it as default.
In the beginning the model = ... generates random function weights w_k and a random bias b_k for every node.

- in split, works for: all?
- in model = ..., works for: MLPClassifier, other?

test_size
We split our data set into training and testing part. The percentage of data points reserved for the testing part can be specified in the test_size. We usually prefer a bigger part of the data set to be in the training part (is this true and why? why not?)

- works for: all?

what about the features???

R2-Score

Bewertet die Leistung eines Schätzers

$$R2(D,f) = 1 - \frac{MSE(D,f)}{MSE(D,f_0)}$$

- f_0: die Konstantfunktion, welche immer $\frac{1}{2}\sum_i y_i$ (Durchschnittspreis) ausgibt
- D: a dataset
- MSE: Verlustfunktion, Mean Squared Error (wie sehr erratene werte von geschätzten werten abweichen) $MSE(D,f) = \frac{1}{n}\sum_{i \leqslant n}(y_i - \hat{y_i})^2$, where $\hat{y_1} = f(x_i)$

ML Models

There are two types of ML Algorithms that can be trained with data:

Regression
→ how many % is something an is.

Linear Regression:
Equation: $y = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + b$
where x is a col in X and b is the bias?
Import: `from sklearn.linear_model import LinearRegression`

Decision Tree Regressor:
Import: `from sklearn.tree import DecisionTreeRegressor`

Neural Network Regressor:
Import: `from sklearn.neural_network import MLPRegressor`

Classification
→ is or isn't.

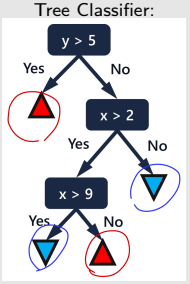
Logistic Regression:
Equation: uses a sigmoid:
$y = \sigma(w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n)$
Import: `from sklearn.linear_model import LogisticRegression`

Decision tree Classifier:
Import: `from sklearn.tree import DecisionTreeClassifier`

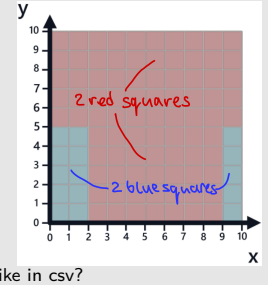
Neural Network Classifier:
Import: `from sklearn.neural_network import MLPClassifier`

Visual Example of Classifier Tree

Tree Classifier:



Data:



how would this data look like in csv?

Plotting the Classifier Tree

if your model is a Classifier Tree you can plot it as follows:

```python
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))
plot_tree(model, feature_names = X.columns)
plt.savefig("./cx_out/tree.png", dpi=300)
```

Plotting the Decision Boundry of Model

although I have no clue what a decision boundry is:

```python
Plot the decision boundary of your model.

plot_decision_boundary(model1, X.values, y.values,
"Model_1", "model1.png")
```

model1: the name of my trained Model
Model 1: the name that will be printed in the png picture, can be chosen freely.

ML with cross validation (grid search)

Can be used to find out the optimal amount of hidden Neuron layers of the depth of a Tree.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split,
GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier,
plot_tree
from sklearn.metrics import accuracy_score
from utils import plot_decision_boundary, plot_data
warnings.filterwarnings("ignore", message="X_has
feature_names,_but_MLPClassifier_was_fitted_without
feature_names")

#read in the data

df = pd.read_csv("data.csv")
X = df[["size", "bright"]].to_numpy()
y = df['label'].to_numpy()

#split the data into training and testing

X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.5, random_state=42)
#random_state controls the randomness???

#Crossvalidate the different tree lengths using
#the train data

dt = DecisionTreeClassifier()
grid = {"max_depth" : range(1, 10)}
#tests from tree length 1 to length 9?
grid = GridSearchCV(dt, param_grid=grid, cv=5)
#cv: crossvalidation: how many shuffle pieces?
grid.fit(X_train, y_train)

best_dt = grid.best_estimator_
#gives best performing tree length

y_pred_train = best_dt.predict(X_train)
y_pred_test = best_dt.predict(X_test)
acc_train = accuracy_score(y_train, y_pred_train)
acc_test = accuracy_score(y_test, y_pred_test)

print("Accuracy_on_train_data:_{:.2f}".format(acc_train))
print("Accuracy_on_test_data:_{:.2f}".format(acc_test))
```

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPRegressor
#split data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
#create grid
param_grid = {'hidden_layer_sizes': [(128,),(64,32),(64,128,32)],
              'activation': ['relu','logistic']}
model = MLPRegressor() #select model
grid_search = GridSearchCV(model, param_grid, cv = 5) #define GridSearchCV
model

grid_search.fit(X_train,y_train) #perform grid search with cross-validation

y_pred = grid_search.predict(X_test) #predict using best parameters
```

.to_numpy: allows me to choose specific columns

random_state = any number: usually train_test_split shuffles the data points randomly every time you call the function. But if you set a random state you freeze it to that specific random state which can be helpful for comparison.

Hyperparameters

Hyperparameters are things you can play around with to adjust model complexity:

1. **NN**: hidden layers, activation functions
2. **Decision tree classifiers**: max depth ???what about regressors???

ML: dealing with non-numeric data

Original data:

Class	Sex	Age	Survived?
Crew	F	Adult	Y
Crew	F	Adult	Y
First	M	Adult	N
First	M	Child	Y
Second	F	Adult	N
Second	M	Child	Y
Second	M	Adult	N

1) Ordinal Encoding:

Class	Sex	Age	Survived?
1	1	1	Y
1	1	1	Y
2	2	1	N
2	2	2	Y
3	1	1	N
3	2	2	Y
3	2	1	N

2) Mean Encoding:

Class	Sex	Age	Survived?
1.0	0.66	0.4	Y
1.0	0.66	0.4	Y
0.5	0.5	0.4	N
0.5	0.5	1.0	Y
0.33	0.66	0.4	N
0.33	0.5	1.0	Y
0.33	0.5	0.4	N

3) One-hot Encoding:

Class = Crew	Class = First	Class = Second	Sex = F	Sex = M	Age = Child	Age = Adult	Survived ?
1	0	0	1	0	0	1	Y
1	0	0	1	0	1	0	Y
0	1	0	0	1	0	1	N
0	1	0	0	1	1	0	Y
0	0	1	1	0	0	1	N
0	0	1	0	1	1	0	Y
0	0	1	1	0	0	1	N

a completely numeric data set

Pro's and Con's

Ordinal Encoding

Pro: Du kannst auch Eigenschaften einer Zahl zuordnen (1: Crew, 2: First ...)

Con: Die Zuordnung kann aber eine "Falsche Nähe" beschreiben. 2 ist näher zu 1 als 3 zu 1, wobei 1 (Crew Memder) logischer eher näher an 3 (economy class) sein sollten.

Mean Encoding

Pro: Versucht die anscheinende Nähe auszugleichen?

Con: Benötigt für ihre Berechnungen Rechenleistung.
Die Resultatspalte "sickert durch" und verändert die Daten, wodurch es Gefahr für Auswendiglernen (Overfitting) entsteht.
Schätzer kann fehlerhafte Rückschlüsse ziehen, anstatt Muster in den (nicht mehr) unabhängigen Daten zu erkennen

One-hot Encoding

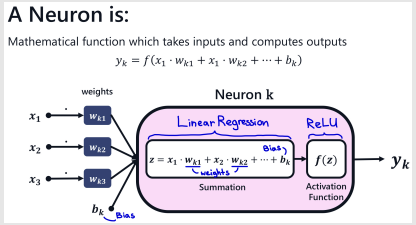
Pro: Datenset ist genau (hohe präzision), da etwas entweder zutrifft oder nicht.

Con: Jede Eigenschaft braucht eine neue Spalte was viel Speicherplatz benötigt.

Neural Network

Standard Neural Network

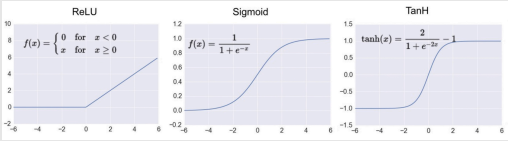
Suited for working with data and tables.



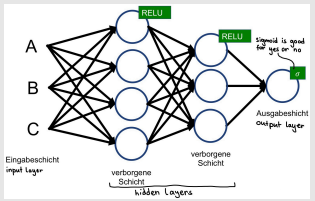
A Neuron is the result of applying an activational function to a linear function.

Activation Function:

- ReLU: $\text{ReLU}(x) = \begin{cases} x & \text{falls } x > 0, \\ 0 & \text{andernfalls} \end{cases}$
- sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent: $tanh(x) = \frac{2}{1+e^{-2x}} - 1$



Linear Function: LinearRegression or LogisticRegression?
Neuron: $ReLu(f(x))$
It is tradition that all of the hidden neuron layers use ReLU and the last Layer uses the Sigmoid activation function.



Convolutional neural Network

Suited for working with images.

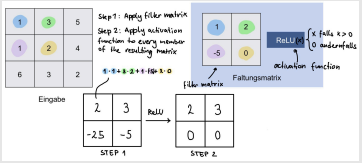
A Neuron is the result of using a convolutional filter and a pulling matrix on the original image matrix.

convolutional filter: consists of filter (tiny 2×2 Image/Matrix) and an activation function (ex: ReLU)
pulling layer: consist of a small pulling matrix with an aggregation operator (MAX)

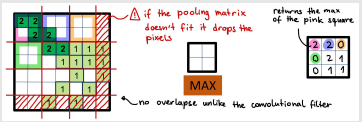
Process:

1. apply filter matrix on original image matrix
2. apply ReLU function on every member of the resulting matrix
3. top it off by applying the pulling matrix

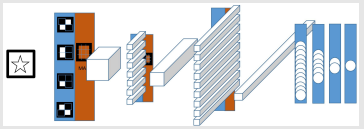
applying convolutional filter:



applying pulling matrix:

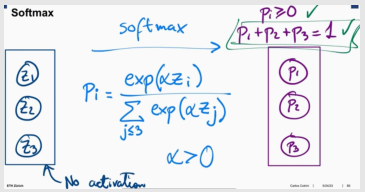


After applzing the pooling matrix we stack all of our output on top of each other. We repeat the process over and over until we recieve a very thin long slice (an array?) that we can work on with a standard neural network:



Softmax

it's an extra function that you apply at the end of your neural network to find out how many percent of something the picture is. For that you make one last layer without an activation function where the amount of neurons stands for the amount of objects you are differentiating between.



Functions Tricks

Import Math

Import the library:

```
import math
```

Convert Degrees to Radians:

```
degrees = 90
radians = math.radians(degrees)
```

ord() function

ord('a') returns the ASCII order of a char

One Line Filter

```
rnum += 1 + (n.left.family if n.left else 0) #example one

a if condition1 else b if condition2 else c #example two
```

sort() function

Sorts the list in ascending order by default. Descending order can be achieved by doing:

```
cars = ['Ford', 'BMW', 'Volvo']

cars.sort(reverse=True)
```

A list of dictionairies sorted based on the "year"value of the dictionairies:

```
# A function that returns the 'year' value:
def myFunc(e):
  return e['year']

cars = [
  {'car': 'Ford', 'year': 2005},
  {'car': 'Mitsubishi', 'year': 2000},
  {'car': 'BMW', 'year': 2019},
  {'car': 'VW', 'year': 2011}
]

cars.sort(key=myFunc
```

other functions may be used:

```
# A function that returns the length of the value:
def myFunc(e):
  return len(e)

cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']

cars.sort(reverse=True, key=myFunc)
```

get() functions

The get() method returns the value of the item with the specified key.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.get("model")

print(x)
```

Slicing

Limiting Floats to x Decimal Points

When wanting to return (ex: print) a long float as a string with a certain amount of numbers after the dot.

```
return "({:.2f},{:.2f})".format(self.latitude, self.longitude)

print("{:.2g}".format(my_float))
```

Pass Continue Break

Pass:
Pass is used to pretend like code was executed. If there is a pass inside of an if statement then it will act as if the if was executed which consequently leads to the else and the elifs after not to be executed.

Continue:
The continue statement is used within loops (such as for or while loops) to skip the rest of the current iteration and move to the next iteration. Like so:

```
for i in range(1, 6):
    if i == 3:
        continue  # Skip iteration when i is 3
    print(i, end=',')
print("\b\b") # Remove the trailing comma and space
#output: 1, 2, 4, 5
```

Break:
Break terminates the loop it is in alltogether.

Programming Concepts

Compiled vs Interpreted

Compiled (C++):
- Program code is translated to assembly.
- Assembly is executed.
- Single translation, with optimizations.
- Usually, higher performance

Interpreted (Python):
- Program code executed together with translation.
- Translation is repeated each time.
- Quick and easy to make minor changes.

Static vs Dynamically Typed

C++ is statically typed:
- Each element has a type defined by the programmer.
- Types used fitting together correctly is checked at compilation, yielding compile time errors (happen during the program itself) if wrong.

Python is dynamically typed:
- Elements have no type in advance.
- At runtime the type is chosen.
- Type changeable at runtime.
- Depending on the type when executing, there may be runtime errrors (happen during the program).
- Errors are more difficult to debug, do not happen all the time.

Generic Programming

The goal of generic programming is to make code as widely usable as possible (no need for new functions for different types).
Can be done with templates in C++.
No need to do anything in Python thanks to dynamic typing.

Functional Programming

The central idea is to pass functions as parameters to functions. An example where we pass a function to "map" is depicted below.

Lambda Functions

Lambda functions are small functions without a specific name, useful to pass into a function as a parameter.

lambda arguments : expression
- Lambda with one argument:

```
n = [1,2,3,4,5]
sqrd_numbers = map(lambda x : x**2, n)
print(list(sqrd_numbers) #[1,4,9,16,25]
```

- Lambda with multiple arguments:

```
y = lambda x,y: x*y
y(5,3) #15
```

Examples of functions that accept functions

- map(func, it) – applies a function on each element of a container.

```
n = [1,2,3,4,5]
sqrd_numbers = map(lambda x : x**2, n)
print(list(sqrd_numbers)#[1,4,9,16,25]
```

- filter(func, it) – removes any elements that don't fulfil a condition.

```
n = [1,2,3,4,5]
even_numbers = filter(lambda x : x%2==0, n)
print(list(even_numbers) #[2,4]
```

- reduce(func,it) – recursively reduce a container to a single value by applying a function to two elements.

```
from functools import reduce
n = [1,2,3,4,5]
sum_numbers = reduce(lambda x,y: x + y, n)#15
```

Containers

Types

Python containers can be divided into ordered (sequences) and unordered (collections) containers.
Sequences include tuple (all types), list (all types), range (integers) and str (characters)
Collections are e.g.: set (non-associative) and dictionary (associative)

Sequences(ordered containers)

- Tuple with 4 Elements:

```
t = (ä", 0, -6, 3.3)
#Tuple with 1 Element: t = ('a',)
```

- Range with 4 elements:

```
r = range(0, 8, 2) r − > 0 2 4 6,
#range(start, stop, step)
```

IMPORTANT: only list is mutable; tuple, range and string are immutable!

General sequences operations

```
print(l[2]) output: hi
```

- Enumeration:

```
enumerate(iterable, start)
#iterable = iterable container (sequence)
#start (optional) = (optional) enumerate starts counting at this number, starts at 0 when omitting start− >
#enumerate(iterable)
#the enumerate(iterable, start) function returns a
#tuple: (index, object).
```

- Enumeration example:

```
  for index, value in enumerate(l):
      print(index, value)
```

- Combine sequences s1 and s2 (zip):

```
z = zip(s1,s2)
```

- Output with a for loop:

```
for name, age in z:
print(name,"− >",age)
```

- Slicing (partial sequence) of a sequence s:

```
partseq = s[start:stop:step]
```

Common list operations

- Add item at the end / Remove item at location i:

```
l.append(value)       del l[i]
```

- Reverse list:

```
l.reverse()
```

- Create a list of k elements with value v:

```
l=[v]*k
```

- To convert a string s to a list of words:

```
s.split(seperator, maxsplit)
#seperator and maxsplit are optional
```

List Comprehension

- Apply a function f(x) to all items in list l:

```
l2 = [f(x) for x in l] #z.g. 2*x for f(x)
```

- Apply a function f(x) to a range:

```
r2 = [f(x) for x in range(1,6)]
```

- Apply a function f(x) only to items in list l that satisfy g(x) (filter):

```
l3 = [f(x) for x in l if g(x)]
```

Common String Operations

- Remove whitespace at beginning and end:

```
s=s.strip()
```

- Convert a string into a list of chars:

```
s = list(s)
```

- Example: check if s is a string with content:

```
type(s) == str and len(s.strip())#False if empty
```

Set operations

- Add/Remove item:

```
s.add(69)       s.remove(29)
```

- Search for an item:

```
12 in S#returns a bool
```

Dictionary Operations

- Delete item:

```
del d["Mortis"] delete item
```

- Make two lists into one dictionary:

```
d2 = dict(zip(cities,code)) #dictionary D2
```

Iterating over a Dictionary

- Iterate over the keys of a dictionary:

```
  for key in d.keys():
      print(key)#Lea Tim Mortis
```

- Iterate over entries of a dictionary:

```
  for item in d.items():
      print(item)#("Lea",22) ("Tim",19) ("Mortis", 69)
```

- Iterate over entries, with keys and values separated:

```
  for key, value in d.items():
      print(key++value)#Lea 22 Tim 19 Mortis 69
```

- Iterate over the values of the dictionary:

```
  for value in d.values():
      print(value) 22 19 69
```

Dictionary/Set Comprehension

- Transform a set into a dictionary by applying f(x) and g(x) on every element in the set to obtain key and value, respectively:

```
d3 = f(x):g(x) for x in s s being a set
```

- Transform a set into a dictionary, only if the element satisfies h(x):

```
d4 = f(x):g(x) for x in s if h(x)
```

- Dictionary comprehension with multiple variables:

```
d5 = f(x):g(y) for x, y in h(z)
#h must return a list of tuples, e.g.: zip, d.items. In the case of d.items, we are applying f(x) on the keys and g(y) on the values of the dictionary.
```

Bemerkungen

Diese ZF wurde für die Vorlesung *Informatik II* für MAVT erstellt. Der Inhalt wurde teils aus der Zusammenfassung *Informatik II* von jlotzer und dsteinhauser übernommen.
Es ist möglich, dass die Zusammenfassung noch Fehler enthält. Ich übernehme keine Haftung für diese. Falls ihr einen Fehler entdeckt oder Verbesserungsvorschläge

habt, könnt ihr euch gerne bei mir melden. Viel Spass beim Coden!

Vasilisa Kirsanova
vkirsanova@student.ethz.ch