Algorithmen zur Bestimmung der Levenshtein-Distanz von Zeichenketten (3./4. Semester) - Natural Language Understanding (3/15) 25 min.

Dr. Ricardo Usbeck
https://github.com/RicardoUsbeck/NLU

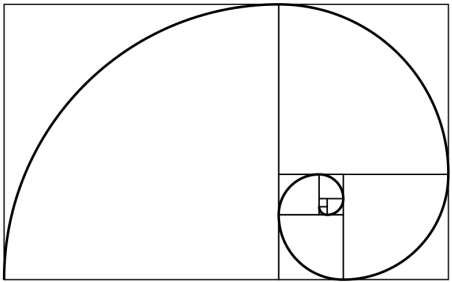
13.07.2020

Wiederholung 1/2: Dynamische Programmierung

- Datenstrukturen und effiziente Algorithmen II (Bellman, 1957)
- Dynamische Programmierung ist eine Optimierungstechnik

Wiederholung 1/2: Dynamische Programmierung

- Datenstrukturen und effiziente Algorithmen II (Bellman, 1957)
- Dynamische Programmierung ist eine Optimierungstechnik
- Bekanntes Problem: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
  - Kann man exponentiell oder polynomiell implementieren
  - Lösung: Bottum-Up, d.h. Teilprobleme speichern



Wiederholung 2/2: Dynamische Programmierung

Dynamische Programmierung Schritte (Nach Cormen et al.):

1. Charakterisiere Struktur der optimalen Lösung
2. Definiere den Wert einer optimalen Lösung rekursiv
3. Berechne den Wert der optimalen Lösung
4. Wir konstruieren eine zugehörige optimale Teillösung aus bereits berechneten Daten

Wiederholung 2/2: Dynamische Programmierung

Dynamische Programmierung Schritte (Nach Cormen et al.):

1. Charakterisiere Struktur der optimalen Lösung
2. Definiere den Wert einer optimalen Lösung rekursiv
3. Berechne den Wert der optimalen Lösung
4. Wir konstruieren eine zugehörige optimale Teillösung aus bereits berechneten Daten

Herangehensweisen für Probleme mit überlappenden aber unabhängigen Teilproblemen:

1. Rekursive Berechnung (Brute-Force/nicht Dynamische Programmierung)
2. Memoization bei der rekursiven Variante hilft bei der Reduktion der Laufzeit
3. Bottum-up Lösung des Problems durch Iteration aller optimalen Teilprobleme
4. Möglichkeit zur Einsparung von Speicherplatz

Motivation: Levenshtein-Distanz/Minimum-Edit-Distance

- Löst das String-Ähnlichkeitsproblem

Motivation: Levenshtein-Distanz/Minimum-Edit-Distance

- Löst das String-Ähnlichkeitsproblem
- Beispiele
  - Rechtschreibkontrolle: graffe $\rightarrow$ graf, graft, grail, giraffe
  - Information Retrieval (z.B. Auto-Complete)
  - Natural Language Understanding (z.B. Kandidatensuche)
  - Machine Translation Systemen zum Finden von Alignments in parallelen Korpora
  - eHumanities zur Annotationsanalyse/Interrater Agreement Bestimmung
  - Nukleotidsequenzähnlichkeit (AATCCGCTAG $\rightarrow$ AAACCCTTAG)

Intuition: Levenshtein-Distanz/Minimum-Edit-Distance

Definition der Operationen und ihrer Kosten um von String A zu String B zu kommen:

- replace $= 1$, wenn Character i, j unterschiedlich sonst replace $= 0$
- delete $= 1$
- insert $= 1$

Intuition: Levenshtein-Distanz/Minimum-Edit-Distance

Definition der Operationen und ihrer Kosten um von String A zu String B zu kommen:

- replace $= 1$, wenn Character i, j unterschiedlich sonst replace $= 0$
- delete $= 1$
- insert $= 1$

Beispiel: Wieviele Operationen braucht man um von 'apfel' zu 'pferd' zu kommen?

Intuition: Levenshtein-Distanz/Minimum-Edit-Distance

Definition der Operationen und ihrer Kosten um von String A zu String B zu kommen:

- replace = 1, wenn Character i, j unterschiedlich sonst replace = 0
- delete = 1
- insert = 1

Beispiel: Wieviele Operationen braucht man um von 'apfel' zu 'pferd' zu kommen?

1. Lösche 'a'
2. Füge 'r' ein
3. Ersetze 'l' durch 'd'

A =apfel, B =pferd $edit_{A,B}(5,5) \Rightarrow 3$

Intuition: Levenshtein-Distanz/Minimum-Edit-Distance

- Betrachte die beiden ersten Character und ignorieren erstmal den Substring, folgende Situation:
  1. Characters sind gleich, tue nichts, kostet nichts (Antwort zum Teilproblem ist Antwort auf dieses Problem)
  2. Characters sind **nicht** gleich, entferne beide - replace, (Antwort zum Teilproblem ist Antwort auf dieses Problem)
  3. Characters sind **nicht** gleich, entferne von A_i - delete (löst Teilproblem für A weiter)
  4. Characters sind **nicht** gleich, füge B_i bei A_i - insert (löst Teilproblem für B weiter)

Intuition: Levenshtein-Distanz/Minimum-Edit-Distance

- Betrachte die beiden ersten Character und ignorieren erstmal den Substring, folgende Situation:
  1. Characters sind gleich, tue nichts, kostet nichts (Antwort zum Teilproblem ist Antwort auf dieses Problem)
  2. Characters sind **nicht** gleich, entferne beide - replace, (Antwort zum Teilproblem ist Antwort auf dieses Problem)
  3. Characters sind **nicht** gleich, entferne von A_i - delete (löst Teilproblem für A weiter)
  4. Characters sind **nicht** gleich, füge B_i bei A_i - insert (löst Teilproblem für B weiter)
- (!) Wichtig, mit jedem Schritt bewegen wir uns weiter in einem der Teilprobleme

Definition: Levenshtein-Distanz/Minimum-Edit-Distance

- Gegeben zwei Strings: A mit Länge n und B mit Länge m

Definition: Levenshtein-Distanz/Minimum-Edit-Distance

- Gegeben zwei Strings: A mit Länge n und B mit Länge m
- Definiere $edit(i, j)$ als die Distanz zwischen $A_{1,\ldots,i}$ und $B_{1,\ldots,j}$ (erste i / j Zeichen)
- Definiere $edit(n, m)$ als Levenshtein-Distanz zwischen A und B

Definition: Levenshtein-Distanz/Minimum-Edit-Distance

- Gegeben zwei Strings: A mit Länge n und B mit Länge m
- Definiere $edit(i, j)$ als die Distanz zwischen $A_{1,\ldots,i}$ und $B_{1,\ldots,j}$ (erste i / j Zeichen)
- Definiere $edit(n, m)$ als Levenshtein-Distanz zwischen A und B
- $edit(0, j) = |j|$ und $edit(i, 0) = |i|$

Definition: Levenshtein-Distanz/Minimum-Edit-Distance

- Gegeben zwei Strings: A mit Länge n und B mit Länge m
- Definiere $edit(i, j)$ als die Distanz zwischen $A_{1,\ldots,i}$ und $B_{1,\ldots,j}$ (erste i / j Zeichen)
- Definiere $edit(n, m)$ als Levenshtein-Distanz zwischen A und B
- $edit(0, j) = |j|$ und $edit(i, 0) = |i|$

$$
\begin{aligned}
edit(i,j) = min(& \\
& cost_{replace} + edit(i-1, j-1), \\
& cost_{delete} + edit(i-1, j), \\
& cost_{insert} + edit(i, j-1))
\end{aligned}
\tag{1}
$$

Definition: Levenshtein-Distanz/Minimum-Edit-Distance

- Gegeben zwei Strings: A mit Länge n und B mit Länge m
- Definiere $edit(i, j)$ als die Distanz zwischen $A_{1,\ldots,i}$ und $B_{1,\ldots,j}$ (erste i / j Zeichen)
- Definiere $edit(n, m)$ als Levenshtein-Distanz zwischen A und B
- $edit(0, j) = |j|$ und $edit(i, 0) = |i|$

$$
\begin{aligned}
edit(i,j) = min(& \\
& cost_{replace} + edit(i-1, j-1), \\
& cost_{delete} + edit(i-1, j), \\
& cost_{insert} + edit(i, j-1))
\end{aligned}
\tag{1}
$$

In Worten: Minimale Anzahl an Operationen, um A nach B zu transformieren

Levenshtein-Distanz/Minimum-Edit-Distance (Rekursion)

$$edit(i,j) = min(cost_{replace} + edit(i-1,j-1), cost_{delete} + edit(i-1,j), cost_{insert} + edit(i,j-1),)$$

```
def edit(a,b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    cost = 1 if a[-1]!= b[-1] else 0
    return min( edit(a[:-1],b[:-1]) + cost,
                edit(a,b[:-1]) +1,
                edit(a[:-1],b) +1)
```

Levenshtein-Distanz/Minimum-Edit-Distance (Rekursion)

$$edit(i,j) = min(cost_{replace} + edit(i-1,j-1), cost_{delete} + edit(i-1,j), cost_{insert} + edit(i,j-1),)$$

```
def edit(a,b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    cost = 1 if a[-1]!= b[-1] else 0
    return min( edit(a[:-1],b[:-1]) + cost,
                edit(a,b[:-1]) +1,
                edit(a[:-1],b) +1)
```

- https://colab.research.google.com/drive/1kZ7BP9OZ9Z2WSTcJrGGKcBpNLBXfWErt#scrollTo=gN78e6up20he&line=1&uniqifier=1

Levenshtein-Distanz/Minimum-Edit-Distance (Rekursion)

$$edit(i,j) = min(cost_{replace} + edit(i-1,j-1), cost_{delete} + edit(i-1,j), cost_{insert} + edit(i,j-1),)$$

Levenshtein-Distanz/Minimum-Edit-Distance (Rekursion)

```
import time

def edit(a,b):
    if len(a) == 0:   return len(b)
    if len(b) == 0:   return len(a)
    cost = 1 if a[-1]!= b[-1] else 0
    return min( edit(a[:-1],b[:-1]) + cost,
                edit(a,b[:-1]) +1,
                edit(a[:-1],b) +1)

start = time.time()
print(edit('',''))
print(edit('pf','pf'))
print(edit('pf','pfer'))
print(edit('aaapf','pfaa'))
print(edit('apfel','pferd'))
print(edit('execution','inception'))
ende = time.time()
print('{:5.3f}s'.format(ende-start))
```

```
0
0
2
4
3
5
0.802s
```

Levenshtein-Distanz/Minimum-Edit-Distance (Rekursion)

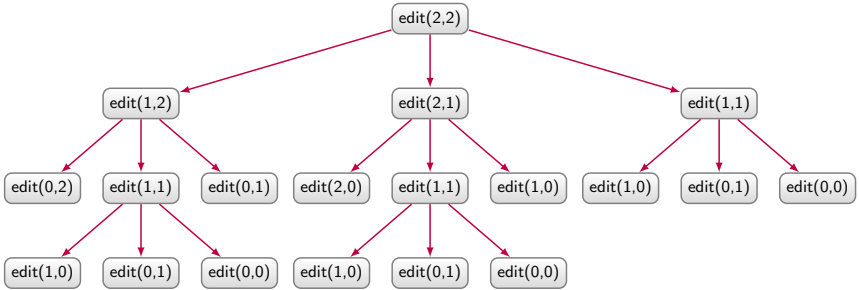
$$edit(i,j) = min(cost_{replace} + edit(i-1,j-1), cost_{delete} + edit(i-1,j), cost_{insert} + edit(i,j-1),)$$

```
def edit(a,b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    cost = 1 if a[-1]!= b[-1] else 0
    return min( edit(a[:-1],b[:-1]) + cost,
                edit(a,b[:-1]) +1,
                edit(a[:-1],b) +1)
```

- https://colab.research.google.com/drive/1kZ7BP9OZ9Z2WSTcJrGGKcBpNLBXfWErt#scrollTo=gN78e6up20he&line=1&uniqifier=1
- Laufzeit $O(3^n)$

Levenshtein-Distanz/Minimum-Edit-Distance (Rekursion)

Rekursionsbaum für $A = ab$ und $B = xy$ als Worst Case.



- Beobachtung: Raum aller möglichen Sequenzen ist sehr groß
- Lösung: Dynamische Programmierung über die Nutzung vorher berechneter, kleinere Teilprobleme von edit(i,j)!

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
a						
p						
f						
e						
l						

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1					
p	2					
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1				
p	2					
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2			
p	2					
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2	3		
p	2					
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2	3	4	
p	2					
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2	3	4	5
p	2					
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2	3	4	5
p	2	1				
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2	3	4	5
p	2	1	2	3	4	5
f	3					
e	4					
l	5					

Levenshtein-Distanz/Minimum-Edit-Distance

$edit(0, j) = |j|$ und $edit(i, 0) = |i|$
$edit(i, j) = min($
$cost_{replace} + edit(i - 1, j - 1),$
$cost_{delete} + edit(i - 1, j),$
$cost_{insert} + edit(i, j - 1))$

		p	**f**	**e**	**r**	**d**
	0	1	2	3	4	5
a	1	1	2	3	4	5
p	2	1	2	3	4	5
f	3	2	1	2	3	4
e	4	3	2	1	2	3
l	5	4	3	2	2	3

Levenshtein-Distanz/Minimum-Edit-Distance

Bekannt als Wagner–Fischer-Algorithmus.

EDIT(A, B)

1 **for** $i \leftarrow 0$ **to** $|A|$
2 **do** $m[i, 0] = i$
3 **for** $j \leftarrow 0$ **to** $|B|$
4 **do** $m[0, j] = j$
5 **for** $i \leftarrow 1$ **to** $|A|$
6 **do for** $j \leftarrow 1$ **to** $|B|$
7 **do if** $A[i] = B[j]$
8 **then** $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]\}$
9 **else** $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]+1\}$
10 **return** $m[|A|, |B|]$

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein-Distanz/Minimum-Edit-Distance

EDIT(A, B)

```
 1  for i ← 0 to |A|
 2  do m[i, 0] = i
 3  for j ← 0 to |B|
 4  do m[0, j] = j
 5  for i ← 1 to |A|
 6  do for j ← 1 to |B|
 7       do if A[i] = B[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|A|, |B|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost1), copy (cost 0)

Levenshtein-Distanz/Minimum-Edit-Distance

EDIT(A, B)

```
 1  for i ← 0 to |A|
 2  do m[i, 0] = i
 3  for j ← 0 to |B|
 4  do m[0, j] = j
 5  for i ← 1 to |A|
 6  do for j ← 1 to |B|
 7      do if A[i] = B[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|A|, |B|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein-Distanz/Minimum-Edit-Distance

EDIT(A, B)

```
 1  for i ← 0 to |A|
 2  do m[i, 0] = i
 3  for j ← 0 to |B|
 4  do m[0, j] = j
 5  for i ← 1 to |A|
 6  do for j ← 1 to |B|
 7       do if A[i] = B[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|A|, |B|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein-Distanz/Minimum-Edit-Distance

EDIT(A, B)

1 **for** $i \leftarrow 0$ **to** $|A|$
2 **do** $m[i, 0] = i$
3 **for** $j \leftarrow 0$ **to** $|B|$
4 **do** $m[0, j] = j$
5 **for** $i \leftarrow 1$ **to** $|A|$
6 **do for** $j \leftarrow 1$ **to** $|B|$
7 **do if** $A[i] = B[j]$
8 **then** $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]\}$
9 **else** $m[i, j] = \min\{m[i\text{-}1, j]+1, m[i, j\text{-}1]+1, m[i\text{-}1, j\text{-}1]+1\}$
10 **return** $m[|A|, |B|]$

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein-Distanz/Minimum-Edit-Distance

```
def edit(a,b):
    if len(a) < len(b): return edit(b, a)
    a = '␣' + a
    b = '␣' + b
    d = {}
    S, T = len(a), len(b)
    for i in range(S):
        d[i, 0] = i
    for j in range (T):
        d[0, j] = j
    for j in range(1,T):
        for i in range(1,S):
            if a[i] == b[j]:
                d[i, j] = d[i-1, j-1]
            else:
                d[i, j] = min(d[i-1, j], d[i, j-1], d[i-1, j-1]) + 1
    return d[S-1, T-1]
```

Levenshtein-Distanz/Minimum-Edit-Distance

- Speicherbedarf $O(|a| \cdot |b|)$
- Laufzeit $O(|a| \cdot |b|)$
- `https://colab.research.google.com/drive/1kZ7BP9OZ9Z2WSTcJrGGKcBpNLBXfWErt#scrollTo=3_2B4Lin226_&line=2&uniqifier=1`

Levenshtein-Distanz/Minimum-Edit-Distance

Levenshtein-Distanz/Minimum-Edit-Distance (Bottum-Up)

```
def edit(a,b):
    if len(a) < len(b): return edit(b, a)
    a = ' ' + a
    b = ' ' + b
    d = {}
    S, T = len(a), len(b)
    for i in range(S):
        d[i, 0] = i
    for j in range (T):
        d[0, j] = j
    for j in range(1,T):
        for i in range(1,S):
            if a[i] == b[j]:
                d[i, j] = d[i-1, j-1]
            else:
                d[i, j] = min(d[i-1, j], d[i, j-1], d[i-1, j-1]) + 1
    return d[S-1, T-1]

start = time.time()
print(edit('',''))
print(edit('pf','pf'))
print(edit('pf','pfer'))
print(edit('aaapf','pfaa'))
print(edit('apfel','pferd'))
print(edit('execution','inception'))
ende = time.time()
print('{:5.3f}s'.format(ende-start))
```

```
0
0
2
4
3
5
0.001s
```

Levenshtein-Distanz/Minimum-Edit-Distance (Speicher reduzieren)

```
def edit(a, b):
  if len(a) < len(b): return edit(b, a)
  if len(a) == 0:     return len(b)
  if len(b) == 0:     return len(a)
  v0, v1 = [None] * (len(a) + 1) ,  [None] * (len(a) + 1)
  for i in range(len(v0)):
      v0[i] = i
  for i in range(len(a)):
      v1[0] = i + 1
      for j in range(len(b)):
          cost = 0 if a[i] == b[j] else 1
          v1[j + 1] = min(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost)
      for j in range(len(v0)):
          v0[j] = v1[j]

  return v1[len(b)]
```

Levenshtein-Distanz/Minimum-Edit-Distance (Speicher reduzieren)

- Speicherbedarf $O(min(|a|, |b|))$
- Laufzeit $O(|a| \cdot |b|)$
- `https://colab.research.google.com/drive/1kZ7BP9OZ9Z2WSTcJrGGKcBpNLBXfWErt#scrollTo=4AX2m6pk4EwO&line=1&uniqifier=1`

Levenshtein-Distanz/Minimum-Edit-Distance (Speicher reduzieren)

Levenshtein-Distanz/Minimum-Edit-Distance (Bottum-up + Speicherreduktion)

```
def edit(a, b):
  if len(a) < len(b): return edit(b, a)
  if len(a) == 0:     return len(b)
  if len(b) == 0:     return len(a)
  v0, v1 = [None] * (len(a) + 1) ,  [None] * (len(a) + 1)
  for i in range(len(v0)):
      v0[i] = i
  for i in range(len(a)):
      v1[0] = i + 1
      for j in range(len(b)):
          cost = 0 if a[i] == b[j] else 1
          v1[j + 1] = min(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost)
      for j in range(len(v0)):
          v0[j] = v1[j]

  return v1[len(b)]

start = time.time()
print(edit('',''))
print(edit('pf','pf'))
print(edit('pf','pfer'))
print(edit('aaapf','pfaa'))
print(edit('apfel','pferd'))
print(edit('execution','inception'))
ende = time.time()
print('{:5.3f}s'.format(ende-start))
```

```
0
0
2
4
3
5
0.001s
```

Take-Away und Ausblick

Was haben wir gelernt?

- Levenshtein-Distanz in Theorie und Praxis

Ist das jetzt sinnvoll?

- Immer noch industrierelevant: Bspw. Elastic Search, Chatbots, Autocomplete
- Inspiration für neue DL-Architekturen: Gu, J., Wang, C., and Zhao, J. (2019). Levenshtein transformer. In Advances in Neural Information Processing Systems (pp. 11181-11191), `http://papers.nips.cc/paper/9297-levenshtein-transformer.pdf`

Wo können wir weiter üben?

- `http://github.com/RicardoUsbeck/NLU`

Was passiert beim nächsten Mal?

- Memoization, Gewichtete Levenshtein-Distanz
- Language Modeling mit N-Grams
- Schnelle Rechtschreibkorrektur

Quellen

- Cormen, T. H., Leiserson, C. E., Rivest, R., and Stein, C. (2017). Algorithmen-Eine Einführung. Walter de Gruyter GmbH and Co KG.
- Jurasky, D., and Martin, J. H. (2000). Speech and Language Processing: An introduction to natural language Processing. Computational Linguistics and Speech Recognition. Prentice Hall, New Jersey. (`https://web.stanford.edu/~jurafsky/slp3/2.pdf`)
- Levenshtein, V. I. (1966, February). Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady (Vol. 10, No. 8, pp. 707-710).
- Wagner, R. A., and Fischer, M. J. (1974). The string-to-string correction problem. Journal of the ACM (JACM), 21(1), 168-173. (`https://dl.acm.org/doi/pdf/10.1145/321796.321811`)
- Fred J. Damerau: A technique for computer detection and correction of spelling errors. In: Communications of the ACM. Band 7, Nr. 3, März 1964, S. 171–176.
- `http://www.let.rug.nl/kleiweg/lev/`, 24.06.2020, Online Tool
- `https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#Python`, 21.06.2020

Lernvideos/Blended Learning

- https://www.youtube.com/watch?v=HXNhEYqFo0o, 14.06.2020
- https://www.youtube.com/watch?v=MiqoA-yF-0M, 19.06.2020
- https://www.youtube.com/watch?v=OQ5jsbhAv_M, 19.06.2020
- https://www.youtube.com/watch?v=Xxx0b7djCrs, 19.06.2020
- https://www.youtube.com/watch?v=ocZMDMZwhCY, 19.06.2020
- https://www.youtube.com/watch?v=8Q2IEIY2pDU, 21.06.2020
- https://www.youtube.com/watch?v=qp8YwtvS3Uo, 21.06.2020
- https://www.youtube.com/watch?v=xFd5P9nyhTw, 30.06.2020

Danke für Ihre Aufmerksamkeit!

🎓 Lernmaterial (VL, Selbsttests, Übungen, Links zu Jupyter Notebooks...)
http://github.com/RicardoUsbeck/NLU

Welche Fragen haben Sie?

- Wladimir Iossifowitsch Lewenstein war ein russischer Mathematiker, der durch die nach ihm benannte, 1965 erfundene Levenshtein-Distanz bekannt wurde. Er machte 1958 seinen Abschluss an der Lomonossow-Universität und lehrte und forschte anschließend am Moskauer Keldysch-Institut für angewandte Mathematik - Wikipedia. Annectode: Dynamische Programmierung wurde durch Bellman vorgeschlagen, da Dynamic Optimization zu sehr nach Forschung klang.

Quelle: https://www.computerhope.com/people/vladimir_levenshtein.htm

Gewichtete Levenshtein-Distanz

- Passe die Kosten pro Operation basierend auf den jeweiligen Characterpaar an
- Wurde entwickelt um Tippfehler zu korrigieren, bspw. m ist wahrscheinlich mit n verwechselt als mit q
- Darum kann man die Edit-Distanz von m nach n kleiner machen als q
- Dazu braucht man eine Gewichtmatrix!
- Anwendung: Computational Biology - Zwei Nukleotidsequenzen alignen (AGTC vs AGGT)
  - DNA Transformation
  - C$\rightarrow$G niedriges Gewicht, weil wahrscheinlicher als C$\rightarrow$A
- Anwendung: Cultural Analytics - Annotations Analyse/Interrater Agreement