

بسم الله الرحمن الرحيم



نام و نام خانوادگی : سارا خسروزاده

نام استاد : آقای دکتر میرسامان تاجبخش

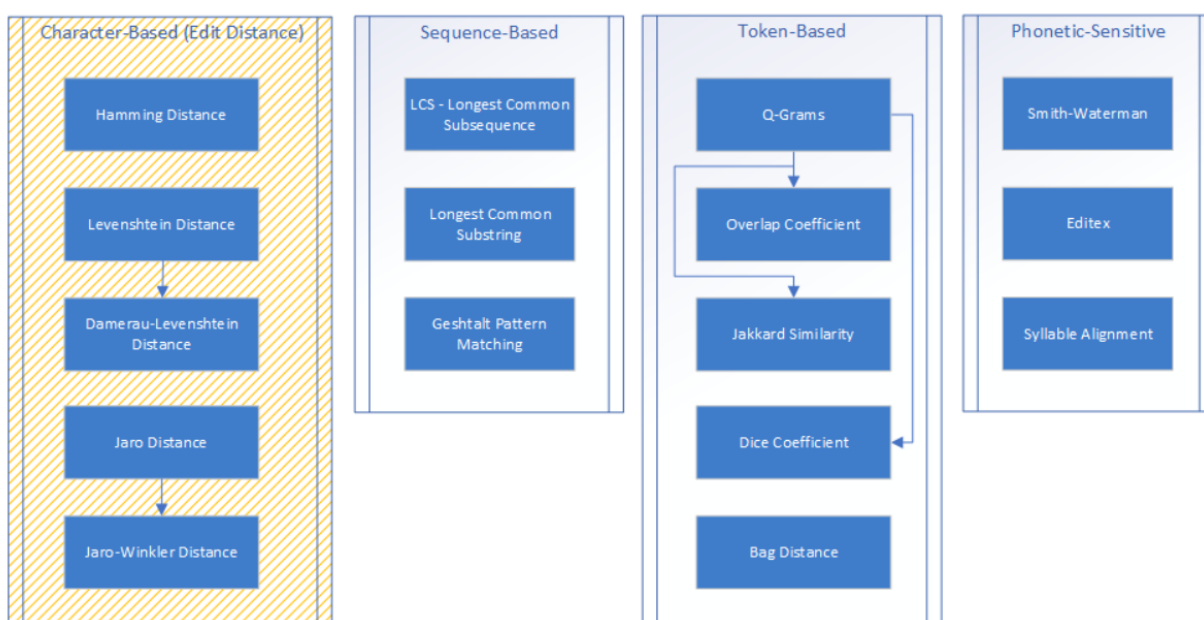
نام درس : بازیابی اطلاعات

مهر 1402

تمرین ۲ - انواع روش‌های edit distance

با توجه به شکل زیر، انواع روش‌های Edit Distance معرفی شده است. هریک را شرح داده و نحوه عملکرد هر کدام را توضیح دهید. اگر کد آماده نیز پیدا شد، آن را نیز معرفی بکنید. برای دو کلمه، با هریک از روش‌ها فاصله را نیز به عنوان مثال حساب نمایید.

هدف از این تمرین ارائه یک white paper در ارتباط با انواع روش‌های Edit Distance می‌باشد. نتیجه گزارش را به عنوان یک فایل PDF آپلود نمایید.



Edit distance به 4 قسمت اصلی تقسیم میشود:

- Character-Based
- Sequence-Based
- Token-Based
- Phonetic-Sensitive

Character-Based

فاصله ویرایش مبتنی بر کاراکتر دو رشته را در سطح کاراکترهای فردی مقایسه می کند . تعداد درجها، حذفها یا تعویضهای مورد نیاز برای تبدیل یک رشته به رشته دیگر را می شمارد و هر کاراکتر را به عنوان یک موجودیت جداگانه در نظر می گیرد. به عنوان مثال، فاصله ویرایش مبتنی بر کاراکتر بین "cat" و "bat" 1 خواهد بود، زیرا یک جایگزین c تا b مورد نیاز است .به طور مشابه، فاصله ویرایش بین "hello" و "helloo" برابر 1 خواهد بود، زیرا اضافه کردن یک "o" اضافی لازم است.

Hamming Distance

فاصله همینگ یک متریک است که برای اندازه گیری اختلاف بین دو رشته با طول مساوی استفاده می شود . در واقع تعداد موقعیت هایی را که عناصر مربوطه در رشته ها متفاوت هستند محاسبه می کند .فاصله همینگ اغلب در علوم کامپیوتر، تئوری کدگذاری و تشخیص خطا استفاده می شود .

برای مثال :

رشته 1: "karate" رشته 2 "karuse" :

برای محاسبه فاصله همینگ بین این دو رشته، کاراکترهای مربوطه را در هر موقعیت با هم مقایسه می کنیم :

در موقعیت 1 'k': در رشته 1 همان 'k' در رشته 2 است .

در موقعیت 2 'a': در رشته 1 همان "a" در رشته 2 است .

در موقعیت 3 'r': در رشته 1 همان 'r' در رشته 2 است .

در موقعیت 4 'a': در رشته 1 با 'u' در رشته 2 متفاوت است .

در موقعیت 5 't': در رشته 1 با 's' در رشته 2 متفاوت است .

در موقعیت 6 'e': در رشته 1 با انتهای رشته 2 متفاوت است، بنابراین به عنوان تفاوت بین رشته ها در نظر گرفته می شود .

بنابراین، فاصله همینگ بین "کاراته" و "کاروسه" 2 است زیرا دو موقعیت وجود دارد که کاراکترهای مربوطه در آنها متفاوت است . به طور کلی، فاصله همینگ را فقط می توان بین رشته هایی با طول مساوی محاسبه کرد . اگر طول رشته ها متفاوت باشد، کاملاً متفاوت در نظر گرفته می شوند و فاصله همینگ برابر با طول رشته بلندتر خواهد بود.

کد مربوطه:

این کد دو رشته را میگیرد و اگر طول آنها برابر نباشد ارور در خروجی چاپ میکند و در صورت برابر بودن طول آنها فاصله همینگ را برمی گرداند.

`zip(str1, str2)` کاراکترهای `str1` و `str2` را از نظر عنصر جفت می کند و دنباله ای از تاپل ها را ایجاد می کند. عبارت `c1 != c2` هر جفت از کاراکترهای `str1` و `str2` را با هم مقایسه می کند و بررسی می کند که آیا برابر نیستند. اگر متفاوت باشند `True` و اگر یکسان باشند `False` را برمی گرداند. تابع `sum()` تمام مقادیر `True` را شمارش کرده و برمی گرداند در نتیجه فاصله بین رشته ها محاسبه میشود.

```
Welcome  hamming.py X
hamming.py > ...
1 def hamming_distance(str1, str2):
2     # Make sure the strings are of equal length
3     if len(str1) != len(str2):
4         raise ValueError("Input strings must have the same length")
5
6     # Calculate Hamming distance
7     distance = sum(c1 != c2 for c1, c2 in zip(str1, str2))
8     return distance
9
10 # Example usage:
11 string1 = "karate"
12 string2 = "karuse"
13
14 result = hamming_distance(string1, string2)
15 print(f"The Hamming distance between '{string1}' and '{string2}' is: {result}")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\مرتاب\7\یاب\اع الطا\یاب\اع الطا\tamrin2> & C:/Users/digiyol.com/AppData/Local/Programs
??????/tamrin2/hamming.py"
The Hamming distance between 'karate' and 'karuse' is: 2
PS C:\مرتاب\7\یاب\اع الطا\یاب\اع الطا\tamrin2>
```

Levenshte in Distance

فاصله لونشتاین تفاوت بین دو رشته را پیدا میکند. به عبارتی دیگر حداقل تعداد ویرایش های تک کاراکتری (درج، حذف یا جایگزینی) مورد نیاز برای تغییر یک رشته به رشته دیگر را محاسبه می کند. و در اینجا لزومی به برابر بودن تعداد کاراکتر های رشته ها نیست.

برای مثال : "kitten" و "sitting".

برای تبدیل "kitten" به "sitting"، می توانیم "k" را با "s" جایگزین کنیم (1 تعویض).

بعد از تعویض، «sitten».

سپس باید "e" را با "i" جایگزین کنیم (1 تعویض).

بعد از تعویض، «sittin».

در نهایت، به "n" "g" را اضافه می کنیم (1 تعویض).

تبدیل نهایی به "sitting" است.

بنابراین، فاصله لונشتاین بین "kitten" و "sitting" 3 است زیرا سه تعویض طول کشید تا رشته اول به رشته دوم تبدیل شود.

کد مربوطه:

این تابع به کمک الگوریتم برنامه نویسی پویا Levenshtein distance بین دو رشته را محاسبه می کند .

ابتدا یک ماتریس به اندازه دو رشته ورودی ایجاد می کنیم .سپس ستون و سطر اول ماتریس را پر می کنیم: اگر رشته اول خالی باشد، فاصله برابر با طول رشته دوم می شود و اگر رشته دوم خالی باشد، فاصله برابر با طول رشته اول می شود.

سپس برای هر خانه داخل ماتریس، فاصله بین دو رشته را به دو روش محاسبه می کنیم :

1. فاصله بین خانه بالای آن و خانه فعلی + 1

```

??????/tamrin2/levenshte.py"
The Levenshtein distance between 'kitten' and 'sitting' is: 3
PS C:\مركز\7\ابى ابى الطاء ابى ابى الطاء ابى ابى الطاء\tamrin2>

```

Damerau-Levenshtein in Distance

فاصله Damerau-Levenshtein اندازه گیری تفاوت بین دو رشته است که به صورت حداقل تعداد عملیات (درج، حذف، جایگزینی یا جابجایی) مورد نیاز برای تغییر یک رشته به رشته دیگر تعریف می شود.

برای مثال :

دو رشته را در نظر بگیریم: "Saturday" and "Sunday"

فاصله Damerau-Levenshtein بین این دو رشته را می توان به صورت زیر محاسبه کرد

در ابتدا، اولین کاراکترهای "S" و "S" مطابقت دارند.

کاراکترهای دوم "a" و "u" متفاوت هستند و نیاز به عملیات جایگزینی دارند . کاراکترهای سوم "t" و "n" نیز متفاوت هستند .

کاراکترهای چهارم "u" و "d" متفاوت هستند .

کاراکترهای باقی مانده "day" و "day" یکسان هستند .بنابراین، تعداد کل عملیات تعویض مورد نیاز برای تبدیل "Saturday" and "Sunday" 3 است.

تفاوت اصلی بین فاصله Damerau-Levenshtein و فاصله Levenshtein این است که فاصله Damerau-Levenshtein امکان عملیات اضافی به نام جابجایی را فراهم می کند.

فاصله Levenshtein حداقل تعداد ویرایش های تک کاراکتری (درج، حذف و جایگزینی) مورد نیاز برای تبدیل یک رشته به رشته دیگر را اندازه گیری می کند. این انتقال دو کاراکتر مجاور را به عنوان یک عملیات معتبر در نظر نمی گیرد .

از طرف دیگر، فاصله Damerau-Levenshtein شامل عملیات جابجایی است. حداقل تعداد عملیات مورد نیاز برای تبدیل یک رشته به رشته دیگر را با اجازه دادن به ویرایش های تک نویسه (درج، حذف، و جایگزینی)، و همچنین جابجایی کاراکترهای مجاور (تعویض دو کاراکتر مجاور) اندازه گیری می کند.

کد مربوطه:

این برنامه یک تابع تحت عنوان "damerau_levenshtein_distance" دارد که از الگوریتم Damerau-Levenshtein برای محاسبه فاصله بین دو رشته استفاده می کند. این الگوریتم، فاصله بین دو رشته را در نظر می گیرد و عملیاتی را انجام می دهد تا یک رشته را به دیگری تبدیل کند. عملیات شامل حذف، درج، جایگزینی و تغییر مکان معکوس دو کاراکتر است.

تابع یک ماتریس به نام "dp" را برای نگهداری فواصل ایجاد می کند. سپس ابتدا سطر و ستون اول ماتریس را مقداردهی اولیه می کند. سپس مقادیر بقیه ماتریس با استفاده از الگوریتم Damerau-Levenshtein محاسبه می شوند و در نهایت فاصله Damerau-Levenshtein در سلول پایین راست ماتریس قرار می گیرد.

برای محاسبه فاصله Jaro، مراحل زیر را دنبال می کنیم:

1. تعداد کاراکترهای منطبق (نویسه ها در موقعیت مشابه) بین دو رشته را بیابید. در این مثال، کاراکترهای منطبق عبارتند از 'a'، 'r'، 'a'، 'm' و 't' 5 کاراکتر مطابق وجود دارد.

2. تعداد جابه جایی ها (نویسه ها در موقعیت های یکسان اما نه مجاور) را با شمارش تعداد دفعاتی که کاراکترهای منطبق نامرتب هستند، بیابید. در این مثال، جابجایی ها 'h' و 't' هستند. 2 جابجایی وجود دارد.

3. فاصله Jaro را با استفاده از فرمول محاسبه کنید:

Jaro Distance = (number of matching characters / length of String 1 + number of matching characters / length of String 2 + (number of matching characters - number of transpositions) / number of matching characters) / 3

The Jaro distance d_j of two given strings s_1 and s_2 is

$$d_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3} \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) & \text{otherwise} \end{cases}$$

Where:

- $|s_i|$ is the length of the string s_i
- m is the number of matching characters
- t is half the number of transpositions

کد مربوطه:

این کد یک تابع برای محاسبه شباهت Jaro بین دو رشته می باشد. شباهت Jaro یک معیار از همسانی بین دو رشته است که بر اساس تطابق حروف و ترتیب آنها در دو رشته تعریف می شود.

تابع `jaro_distance()` دو ورودی `s1` و `s2` دریافت می کند که به ترتیب دو رشته است که می خواهیم شباهتشان را محاسبه کنیم. اگر رشته ها برابر باشند خروجی 1.0 است. در غیر این صورت ، طول رشته ها را محاسبه می کند و حداکثر فاصله مجاز بین مطابقت ها را محاسبه می کند. سپس با استفاده از الگوریتم Jaro ، تعداد مطابقت ها بین دو رشته را محاسبه می کند و تعداد `transpositions` را شمارش می کند. در نهایت ، با استفاده از فرمول محاسبه شباهت Jaro ، شباهت نهایی بین دو رشته را برمی گرداند.

در اینجا بین رشته های `martha` و `marhta` شباهت Jaro محاسبه شده است و نتیجه 0.944444 برگردانده می شود. و وقتی این مقدار را از یک کم میکنیم فاصله jaro را به ما میدهد.

```

Go Run Terminal Help
Welcome hamming.py levenshte.py damerau.py
jaro.py > jaro_distance
1 from math import floor, ceil
2
3 # Jaro Similarity of two s
4 def jaro_distance(s1, s2):
5
6     # If the s are equal
7     if (s1 == s2):
8         return 1.0
9
10    # Length of two s
11    len1 = len(s1)
12    len2 = len(s2)
13
14    # Maximum distance upto which matching
15    # is allowed
16    max_dist = floor((max(len1, len2) / 2) - 1)
17
18    # Count of matches
19    match = 0
20
21    # Hash for matches
22    hash_s1 = [0] * len(s1)
23    hash_s2 = [0] * len(s2)
24
25    # Traverse through the first
26    for i in range(len1):
27
28        # Check if there is any matches
29        for j in range(max(0, i - max_dist),
30                        min(len2, i + max_dist + 1)):
31
32            # If there is a match

```

```
Go Run Terminal Help
Welcome hamming.py levenshte.py damerau.py jaro.py
jaro.py > jaro_distance

31
32     # If there is a match
33     if (s1[i] == s2[j] and hash_s2[j] == 0):
34         hash_s1[i] = 1
35         hash_s2[j] = 1
36         match += 1
37         break
38
39     # If there is no match
40     if (match == 0):
41         return 0.0
42
43     # Number of transpositions
44     t = 0
45     point = 0
46
47     for i in range(len1):
48         if (hash_s1[i]):
49
50             # Find the next matched character
51             # in second
52             while (hash_s2[point] == 0):
53                 point += 1
54
55             if (s1[i] != s2[point]):
56                 t += 1
57                 point += 1
58     t = t//2
```

```

    # Return the Jaro Similarity
    return (match/ len1 + match / len2 +
            (match - t) / match) / 3.0

# Driver code
s1 = "martha"
s2 = "marhta"

# Prjaro Similarity of two s
print(1-round(jaro_distance(s1, s2),6))
```

```
from jarowinkler import *  
print(1-jaro_similarity("martha", "marhta"))  
print(1-jarowinkler_similarity("martha", "marhta"))
```

```
0.05555555555555547  
0.03888888888888886
```

Jaro-Winkler Distance

فاصله Jaro-Winkler یک معیاری است که فاصله ویرایش بین دو دنباله را اندازه‌گیری می‌کند. هر چه فاصله Jaro-Winkler برای دو رشته کمتر باشد، رشته ها شبیه تر هستند. امتیاز به گونه ای نرمال می شود که 0 به معنای تطابق دقیق و 1 به معنای عدم وجود شباهت است.

Jaro-Winkler similarity uses a [prefix](#) scale p which gives more favorable ratings to strings that match from the beginning for a set prefix length ℓ . Given two strings s_1 and s_2 , their Jaro-Winkler similarity sim_w is:

$$sim_w = sim_j + \ell p(1 - sim_j),$$

where:

- sim_j is the Jaro similarity for strings s_1 and s_2
- ℓ is the length of common prefix at the start of the string up to a maximum of 4 characters
- p is a constant [scaling factor](#) for how much the score is adjusted upwards for having common prefixes. p should not exceed 0.25 (i.e. 1/4, with 4 being the maximum length of the prefix being considered), otherwise the similarity could become larger than 1. The standard value for this constant in Winkler's work is $p = 0.1$

The Jaro-Winkler distance d_w is defined as $d_w = 1 - sim_w$.

Although often referred to as a *distance metric*, the Jaro-Winkler distance is not a [metric](#) in the mathematical sense of that term because it does not obey the [triangle inequality](#).^[4] The Jaro-Winkler distance also does not satisfy the identity axiom $d(x, y) = 0 \leftrightarrow x = y$.

برای مثال:

فاصله Jaro-Winkler برای رشته های "apple" و "applet" را می توان به صورت زیر محاسبه کرد :

مرحله 1:

محاسبه فاصله Jaro: فاصله Jaro با شمارش تعداد کاراکترهای مشترک بین دو رشته و تعداد جابجایی های مورد نیاز برای یکسان شدن آنها تعیین می شود. در این مورد، کاراکترهای رایج apple و applet ، با جابجایی 0 هستند. فاصله Jaro 0.05557 است.

مرحله 2: فاکتور Jaro-Winkler را اعمال کنید :

فاصله Jaro-Winkler شامل یک ضریب مقیاس اضافی بر اساس طول میشوند مشترک و یک ثابت از پیش تعریف شده (p) است. با فرض $p = 0.1$ ، می توانیم از فرمول برای تنظیم فاصله Jaro استفاده کنیم

بنابراین، فاصله Jaro-Winkler بین "apple" و "applet" 0.033333 است.

```
from jarowinkler import *  
print(jaro_similarity("apple", "applet"))  
print(jarowinkler_similarity("apple", "applet"))
```

```
0.9444444444444445  
0.9666666666666667
```

```
✓ 0s from jarowinkler import *  
print("jaro distance is :",1-jaro_similarity("apple", "applet"))  
print("jarowinkler distance is:",1-jarowinkler_similarity("apple", "applet"))
```


```
jaro distance is : 0.05555555555555547  
jarowinkler distance is: 0.03333333333333326
```


کد مربوطه: تابع jaro_similarity :

این تابع شباهت Jaro بین دو رشته s1 و s2 را بر اساس فرمول فاصله Jaro محاسبه می کند. این کاراکترهای مشترک را در یک فاصله تطبیق مشخص جستجو می کند و شباهت Jaro را با استفاده از طول کاراکترها و جابجایی های رایج محاسبه می کند .

تابع jaro_winkler_similarity: این تابع شباهت Jaro را با گنجاندن عامل Jaro-Winkler، که شامل ضریب مقیاس بندی p و کاراکترهای تطبیق در ابتدای رشته ها است، گسترش می دهد. شباهت Jaro-Winkler را با تنظیم شباهت Jaro بر اساس طول پیشوند مشترک و ضریب مقیاس بندی محاسبه می کند .

```
jarowinkler.py > [?] string1
1 def jaro_similarity(s1, s2):
2     len_s1, len_s2 = len(s1), len(s2)
3     match_distance = max(len_s1, len_s2) // 2 - 1
4
5     common_chars_s1 = []
6     common_chars_s2 = []
7
8     for i, char in enumerate(s1):
9         start = max(0, i - match_distance)
10        end = min(i + match_distance + 1, len_s2)
11
12        if char in s2[start:end]:
13            common_chars_s1.append(char)
14            common_chars_s2.append(s2[start:end].index(char))
15
16    m = len(common_chars_s1)
17    if m == 0:
18        return 0.0
19
20    transpositions = sum(c1 != c2 for c1, c2 in zip(common_chars_s1, common_chars_s2)) // 2
21    jaro_similarity = (m / len_s1 + m / len_s2 + (m - transpositions) / m) / 3
22    return jaro_similarity
23
24
25 def jaro_winkler_similarity(s1, s2, p=0.1):
26     jaro_sim = jaro_similarity(s1, s2)
27     common_prefix_len = 0
28
29     for i, (c1, c2) in enumerate(zip(s1, s2)):
30         if c1 == c2:
31             common_prefix_len += 1
32         else:
33             break
34
35     jaro_winkler_sim = jaro_sim + (common_prefix_len * p * (1 - jaro_sim))
36     return jaro_winkler_sim
```

```
37 
38 string1 = "apple"
39 string2 = "applet"
40 jw_similarity = jaro_winkler_similarity(string1, string2)
41 print("Jaro-Winkler Distance:", 1-jw_similarity)
42
```

Sequence-Based

فاصله ویرایش مبتنی بر توالی ، ترتیب و توالی کاراکترها را در رشته ها در نظر می گیرد . حداقل تعداد عملیات مورد نیاز برای تبدیل یک دنباله به دنباله دیگر را اندازه گیری می کند، جایی که یک دنباله می تواند هر مجموعه مرتب شده ای از آیتم ها باشد .به عنوان مثال، فاصله ویرایش مبتنی بر دنباله بین [1, 2, 3, 4] و [1, 3, 2, 4] 2 خواهد بود، زیرا برای تطبیق دنباله ها به دو عملیات (تعویض عنصر دوم و سوم) نیاز است.

LCS-longest common subsequence

با توجه به دو رشته S1 و S2 ، وظیفه یافتن طول طولانی ترین زیر دنباله مشترک، یعنی طولانی ترین زیر دنباله موجود در هر دو رشته است .طولانی ترین زیر دنباله مشترک (LCS) به عنوان طولانی ترین زیر دنباله ای تعریف می شود که در همه دنباله های ورودی داده شده مشترک است.

برای مثال:

Input: S1 = "AGGTAB", S2 = "GTXAYB"

Output: 4

Explanation: The longest subsequence which is present in both strings is "GTAB".

کد مربوطه:

تابع lcs از تابع بازگشتی برای محاسبه طول LCS بین رشته های ورودی X و Y با طول های m و n استفاده می کند. اگر هر یک از رشته ها خالی باشد، طول LCS 0 است. اگر آخرین کاراکترهای هر دو رشته مطابقت داشته باشند، 1 به طول LCS اضافه می شود و تابع به صورت بازگشتی برای رشته های فرعی باقی مانده فراخوانی می شود. اگر آخرین کاراکترهای رشته ها مطابقت نداشته باشند، تابع حداکثر طول LCS را می گیرد که با نادیده گرفتن آخرین کاراکتر هر یک از رشته ها به دست می آید. سپس از این تابع برای یافتن طول LCS بین رشته های داده شده استفاده می کند.

```
lcs.py X
lcs.py > ...
1 def lcs(X, Y, m, n):
2     if m == 0 or n == 0:
3         return 0
4     elif X[m-1] == Y[n-1]:
5         return 1 + lcs(X, Y, m-1, n-1)
6     else:
7         return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n))
8
9
10 # Driver code
11 if __name__ == '__main__':
12     S1 = "AGGTAB"
13     S2 = "GXTXAYB"
14     print("Length of LCS is", lcs(S1, S2, len(S1), len(S2)))

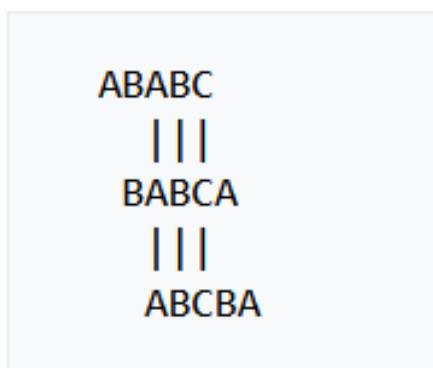
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python/Python311/python.exe "c:/??? 7/??????? ?????/??????? ?????? ??????
Length of LCS is 4
PS C:\... 7) ...\tamin2>
```

Longest common substring

در علوم کامپیوتر، طولانی ترین زیررشته مشترک از دو یا چند رشته، طولانی ترین رشته ای است که زیررشته ای از همه آنهاست. ممکن است بیش از یک رشته فرعی مشترک وجود داشته باشد.

برای مثال:

رشته های "ABABC"، "BABCA" و "ABCBA" تنها دارای طولانی ترین زیررشته مشترک هستند، یعنی "ABC" به طول 3.



Input : X = "GeeksforGeeks", y = "GeeksQuiz"

Output : 5

Explanation:

The longest common substring is "Geeks" and is of length 5.

کد مربوطه:

تابع LCSuff یک آرایه دوبعدی LCSuff را برای ذخیره طول طولانی ترین پسوند مشترک در هر ترکیبی از شاخص های رشته های «X» و «Y» مقداردهی می کند .
همچنین یک نتیجه متغیر را برای ذخیره طول طولانی ترین زیررشته مشترک یافت شده مقداردهی می کند .

تابع از طریق رشته های 'X' و 'Y' تکرار می شود و آرایه LCSuff را بر اساس طول رشته های فرعی مشترک پر می کند. همچنین حداکثر طول یافت شده را پیدا می کند. پس از تکرار، تابع مقدار ذخیره شده در نتیجه را برمی گرداند که نشان دهنده طول طولانی ترین زیررشته مشترک است.

```
1 def LCSuff(X, Y, m, n):
2
3     LCSuff = [[0 for k in range(n+1)] for l in range(m+1)]
4
5     # To store the length of
6     # longest common substring
7     result = 0
8
9     # Following steps to build
10    # LCSuff[m+1][n+1] in bottom up fashion
11    for i in range(m + 1):
12        for j in range(n + 1):
13            if (i == 0 or j == 0):
14                LCSuff[i][j] = 0
15            elif (X[i-1] == Y[j-1]):
16                LCSuff[i][j] = LCSuff[i-1][j-1] + 1
17                result = max(result, LCSuff[i][j])
18            else:
19                LCSuff[i][j] = 0
20    return result
21
22
23    # Driver Code
24    X = 'GeeksforGeeks'
25    Y = 'GeeksQuiz'
26
27    m = len(X)
28    n = len(Y)
29
30    print('Length of Longest Common Substring is',
31          LCSuff(X, Y, m, n))
32
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Length of Longest Common Substring is 5
PS C:\مرت7\یابای\اع الطا یابای\اع الطا یابای\مرت7> |

Geshtalt pattern matching

یک الگوریتم تطبیق رشته برای تعیین شباهت دو رشته است. شباهت دو رشته S_1 و S_2 با فرمول تعیین می شود، دو برابر تعداد کاراکترهای مطابق K_m تقسیم بر تعداد کل کاراکترهای هر دو رشته محاسبه می شود. کاراکترهای منطبق به عنوان طولانی ترین زیررشته مشترک به علاوه تعداد نویسه های منطبق در مناطق غیر منطبق در دو طرف طولانی ترین زیررشته مشترک تعریف می شوند.

$$D_{ro} = \frac{2K_m}{|S_1| + |S_2|}$$

where the similarity metric can take a value between zero and one:

$$0 \leq D_{ro} \leq 1$$

مقدار 1 نشان دهنده تطابق کامل دو رشته است، در حالی که مقدار 0 به این معنی است که هیچ تطابقی و حتی یک حرف مشترک وجود ندارد.

Token-Based

فاصله ویرایش مبتنی بر توکن رشته ها را به واحدهای کوچکتری به نام نشانه ها (کلمات، عبارات یا هر واحد تعریف شده) تقسیم می کند و نشانه ها را به جای کاراکترهای جداگانه با هم مقایسه می کند. حداقل تعداد عملیات ویرایش مورد نیاز برای تبدیل یک مجموعه از توکن ها به دیگری را اندازه گیری می کند. به عنوان مثال، فاصله ویرایش مبتنی بر رمز بین "I love pizza" و "I love pasta" خواهد بود، زیرا یک جایگزین (جایگزینی "پیتزا" با "پاستا") لازم است.

Q-grams

اندازه گیری فاصله رشته بر اساس شمارش تعداد وقوع Q-grams های مختلف در دو رشته است. رشته‌ها هر چه شبیه تر باشند، Q-grams مشترک بیشتری دارند.

Q-gram ها که به عنوان n-gram نیز شناخته می شوند، دنباله های پیوسته ای از آیتم های q کاراکترها، کلمات یا نمادها هستند که از یک رشته معین استخراج می شوند. اصطلاح "q" به اندازه توالی ها اشاره دارد. به عنوان مثال، اگر q برابر 2 باشد، دنباله ها bigrams نامیده می شوند .

برای مثال:

q = 2 (bigrams), the q-grams of "example" and "ample" are:

'am', 'mp', 'pl', 'le'

کد مربوطه:

تابع generate_qgrams روی رشته تکرار می شود، و در هر موقعیت i ، یک رشته فرعی به طول q استخراج می کند و آن را به لیست q-gram ها اضافه می کند.

تابع find_common_qgrams مجموعه ای حاوی q-gram های مشترک بین دو رشته ورودی را برمیگرداند.

مجموعه A: {سیب، موز، پرتقال، انبه} مجموعه B: {موز، انبه، کیوی}

برای محاسبه ضریب Overlap ابتدا نشانه های مشترک بین مجموعه ها را پیدا می کنیم که عبارتند از "موز" و "انبه". با شمارش تعداد کل توکن ها در هر مجموعه، داریم:

$$|A| = 4 \quad |B| = 3$$

اکنون می توانیم ضریب همپوشانی را محاسبه کنیم:

ضریب همپوشانی = (تعداد رشته های مشترک) / مینیمم(تعداد کل رشته ها در هر دو مجموعه)

$$(2) / \min(4, 3) = 2/3$$

بنابراین، ضریب همپوشانی برای این مثال $2/3$ است.

کد مربوطه:

این کد ابتدا دو ست از رشته ها را میگیرد و با استفاده از تابع مقدار Overlap coefficient را محاسبه میکند.

طریقه محاسبه به این صورت است که طبق فرمول بالا ابتدا تعداد رشته های مشترک و سپس مینیمم ست هارا پیدا کرده و مقدار را محاسبه میکند.

```
q_grams.py  overlapCoefficient.py X
overlapCoefficient.py > overlap_coefficient
1 def overlap_coefficient(set1, set2):
2     common_elements = len(set1.intersection(set2))
3     total_unique_elements = min(len(set1), len(set2))
4
5     if total_unique_elements == 0:
6         return 0.0 # Avoid division by zero
7
8     coefficient = common_elements / total_unique_elements
9     return coefficient
10
11 # Example usage:
12 set1 = {"apple", "orange", "banana"}
13 set2 = {"orange", "banana", "grape"}
14
15 result = overlap_coefficient(set1, set2)
16 print(f"Overlap coefficient between Set 1 and Set 2: {result}")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\مرت7\ابای\اطاع\اطاع\اطاع\tamrin2> & C:/Users/digi
c:/??? 7/???????? ????/???????? ????/???/tamrin2/overlapCoefficient.
Overlap coefficient between Set 1 and Set 2: 0.6666666666666666
PS C:\مرت7\ابای\اطاع\اطاع\اطاع\tamrin2> |
```

Jakkard similarity

Jaccard similarity معیاری است که نشان می دهد چگونه دو مجموعه مشابه هستند. در شباهت جاکارد مبتنی بر توکن، ما مجموعه‌هایی از نشانه‌ها را به جای مجموعه‌ای از عناصر در نظر می‌گیریم. نشانه‌ها می‌توانند کلمات، کاراکترها یا هر واحد

متن دیگری باشند که می خواهید مقایسه کنید. سپس شباهت جاکارد بر اساس همپوشانی توکن ها در مجموعه ها محاسبه می شود.

$$J(A,B) = |A \cap B| / |A \cup B|$$

برای مثال :

Set 1={"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}

Set 2={"A", "quick", "brown", "dog", "jumps", "over", "the", "lazy", "fox"}

The intersection of sets Set 1 and Set 2 is

{"quick", "brown", "jumps", "over", "the", "lazy"}

$$J(\text{Set 1, Set 2}) = 0.9$$

```
# Download 'punkt' resource
nltk.download('punkt')

def jaccard_similarity(set1, set2):
    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return intersection / union if union != 0 else 0

def tokenize(text):
    return set(word_tokenize(text.lower())) # Assuming you want case-insensitive comparison

# Example usage
document1 = "The quick brown fox jumps over the lazy dog."
document2 = "A quick brown dog jumps over the lazy fox."

# Tokenize the documents
tokens1 = tokenize(document1)
tokens2 = tokenize(document2)

# Calculate Jaccard similarity
similarity = jaccard_similarity(tokens1, tokens2)

print(f"Jaccard Similarity: {similarity}")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
Jaccard Similarity: 0.9
```

Dice coefficient

Dice coefficient یک معیار تشابه است که معمولاً در تقسیم‌بندی تصویر، پردازش زبان طبیعی و سایر زمینه‌هایی که نیاز به اندازه‌گیری شباهت بین دو مجموعه وجود دارد، استفاده می‌شود.

$$\frac{2 * |X \cap Y|}{|X| + |Y|}$$

برای مثال:

Set 1={"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}

Set 2={"A", "quick", "brown", "dog", "jumps", "over", "the", "lazy", "fox"}

The intersection of sets Set 1 and Set 2 is

{"quick", "brown", "jumps", "over", "the", "lazy"}

$D(\text{Set 1, Set 2}) \approx 0.888$

```
def dice_coefficient(set1, set2):
    intersection = len(set1.intersection(set2))
    dice = (2.0 * intersection) / (len(set1) + len(set2))
    return dice

# Example usage
set1 = {"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}
set2 = {"A", "quick", "brown", "dog", "jumps", "over", "the", "lazy", "fox"}

# Calculate Dice coefficient
dice_coef = dice_coefficient(set1, set2)

print(f"Dice Coefficient: {dice_coef}")
```

Dice Coefficient: 0.8888888888888888

Bag distance

پارامتری برای محاسبه شباهت بین دو رشته است که به صورت زیر بیان می شود:

For two strings X and Y, the Bag distance is:

$$\max(|\text{bag}(\text{string1}) - \text{bag}(\text{string2})|, |\text{bag}(\text{string2}) - \text{bag}(\text{string1})|)$$

برای مثال برای دو رشته زیر مقدار Bag distance این گونه به دست آمده است:

```
('cat', 'hat')=1
```

```
>>> bd = BagDistance()
>>> bd.get_raw_score('cat', 'hat')
1
>>> bd.get_raw_score('Niall', 'Neil')
2
>>> bd.get_raw_score('aluminum', 'Catalan')
5
>>> bd.get_raw_score('ATCG', 'TAGC')
0
>>> bd.get_raw_score('abcde', 'xyz')
5
```

Phonetic-Sensitive

این فاصله ویرایش به جای املای دقیق کلمات، شباهت آوایی بین رشته ها را در نظر می گیرد. این به ویژه در هنگام برخورد با نام ها یا کلماتی که دارای املا یا تغییرات معتبر متعدد هستند مفید است. برای محاسبه این نوع فاصله ویرایش می توان از الگوریتم های آوایی مانند Soundex یا Metaphone استفاده کرد.

Smith-Waterman

الگوریتم اسمیت-واترمن به جای نگاه کردن به کل دنباله، بخش‌هایی را با تمام طول‌های ممکن مقایسه می‌کند و معیار تشابه را بهینه می‌کند. الگوریتم اسمیت-واترمن (همانطور که از نامش پیداست) یک الگوریتم است. مستقل از هر تابع فاصله است. یک تراز را محاسبه می‌کند که هزینه‌های داده شده توسط تابع فاصله معین را به حداقل می‌رساند. هدف آن تراز کردن دو دنباله به گونه‌ای است که دنباله‌های فرعی مشابه با هم تراز شوند. برای مثال:

```
alignment2 = sequence2[j-1] + alignment2
i -= 1
j -= 1
elif current_score == up + gap_penalty:
    alignment1 = sequence1[i-1] + alignment1
    alignment2 = '-' + alignment2
    i -= 1
elif current_score == left + gap_penalty:
    alignment1 = '-' + alignment1
    alignment2 = sequence2[j-1] + alignment2
    j -= 1

return alignment1, alignment2

# Example usage
sequence1 = "AGCACACA"
sequence2 = "ACACACTA"

alignment1, alignment2 = smith_waterman(sequence1, sequence2)
print("Sequence 1:", alignment1)
print("Sequence 2:", alignment2)
```

```
Sequence 1: AGCACAC-A
Sequence 2: A-CACACTA
```

Editex

Editex یک اندازه گیری فاصله آوایی است که ویژگی های فواصل ویرایش را با استراتژی گروه بندی حروف مورد استفاده توسط Soundex و Phonix ترکیب می کند.

برای مثال:

`ed.get_raw_score('aluminum', 'Catalan')=12`

```
editex.py > ...
1 def editex_distance(s1, s2, substitution_cost=1, transposition_cost=1):
2     m, n = len(s1), len(s2)
3
4     # Initialize the distance matrix
5     dp = [[0] * (n + 1) for _ in range(m + 1)]
6
7     # Fill in the matrix using dynamic programming
8     for i in range(m + 1):
9         for j in range(n + 1):
10             if i == 0:
11                 dp[i][j] = j
12             elif j == 0:
13                 dp[i][j] = i
14             else:
15                 cost = 0 if s1[i-1] == s2[j-1] else substitution_cost
16                 dp[i][j] = min(dp[i-1][j] + 1,          # Deletion
17                               dp[i][j-1] + 1,          # Insertion
18                               dp[i-1][j-1] + cost)      # Substitution
19
20                 if i > 1 and j > 1 and s1[i-1] == s2[j-2] and s1[i-2] == s2[j-1]:
21                     dp[i][j] = min(dp[i][j], dp[i-2][j-2] + transposition_cost) # Transposition
22
23     return dp[m][n]
24
25 # Example usage
26 word1 = "kitten"
27 word2 = "sitting"
28
29 distance = editex_distance(word1, word2)
30 print(f"Editex Distance between '{word1}' and '{word2}': {distance}")
31
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Editex Distance between 'kitten' and 'sitting': 3
PS C:\Users\Z\Documents> cd C:\Users\Z\Documents\Projects\Editex; python editex.py

Syllable alignment

فرآیند تطبیق هجاهای متناظر در دو کلمه، اغلب به منظور مقایسه ساختار آوایی آنها یا تجزیه و تحلیل الگوهای تلفظ آنها اشاره دارد. معمولاً در سنجش شباهت آوایی و تشخیص گفتار استفاده می شود.

برای مثال:

فرض کنید می خواهیم هجاهای دو کلمه "water" و "water" را تراز کنیم:

- ****Word 1: water****
- Syllables: wa-ter
- ****Word 2: waiter****
- Syllables: wait-er

در این حالت هجاها را به صورت زیر تراز می کنیم:

wa-ter

wait-er

در اینجا، هر خط نشان دهنده یک جفت هجای تراز شده است. خط تیره (-) تراز را نشان می دهد. این هم تراز می تواند در تحلیل های آوایی مختلف، مانند مقایسه الگوهای تلفظ یا اجرای الگوریتم های حساس به آوایی مفید باشد.

کد مربوطه:

`get_syllables` یک کلمه را به عنوان ورودی می گیرد و فهرستی از رونویسی های آوایی آن را برمی گرداند. از فرهنگ لغت تلفظ CMU برای دریافت رونویسی آوایی برای

کلمه داده شده استفاده می کند. اگر کلمه در فرهنگ لغت یافت شود، رونویسی های آوایی را استخراج کرده و به لیست اضافه می کند.

`align_syllables` دو کلمه را به عنوان ورودی می گیرد و لیستی از جفت هجاهای تراز شده را برمی گرداند. از تابع `get_syllables` برای دریافت رونویسی آوایی برای هر کلمه استفاده می کند. سپس هجاها را با زیپ کردن رونویسی های آوایی متناظر با هم، تا حداقل طول دو `list`، تراز می کند.

```
import nltk
nltk.download("cmudict")
def get_syllables(word):
    syllables = []
    phones = nltk.corpus.cmudict.dict().get(word.lower())
    if phones:
        for phone_list in phones:
            syllables.extend(phone_list)
    return syllables

def align_syllables(word1, word2):
    syllables1 = get_syllables(word1)
    syllables2 = get_syllables(word2)

    min_len = min(len(syllables1), len(syllables2))

    alignment = list(zip(syllables1[:min_len], syllables2[:min_len]))

    return alignment

# Example usage:
word1 = "water"
word2 = "waiter"
alignment_result = align_syllables(word1, word2)

print(f"Syllable alignment between '{word1}' and '{word2}':")
for pair in alignment_result:
    print(f"{pair[0]:<10} {pair[1]:<10}")
```

```
Python 3.7.4 Shell
```

```
Syllable alignment between 'water' and 'waiter':
```

W	W
AO1	EY1
T	T
ER0	ER0

پایان