String distance

Hyunjoong Kim

soy.lovit@gmail.com

github.com/lovit

String distance

- 벡터는 다양한 metric 을 이용하여 거리나 유사도를 정의할 수 있습니다.
  - Euclidean, Cosine, …
- 단어 간 거리는 형태적 거리와 의미적 거리로 정의할 수 있습니다.
  - 의미적으로 비슷한 단어
    - (점심, 저녁) vs (점심, 자동차)
  - 형태적으로 비슷한 단어
    - (서비스, 써비스) vs. (서비스, 소보루)

String distance

- 단어 간 형태적 거리를 정의하기 위해서 다양한 metric 이 제안되었습니다.
  - Levenshtein (Edit) distance
  - Jaro-Winkler
  - Cosine
  - Jaccard
  - Hamming
  - TF-IDF
  - …

String distance

- 오탈자와 정자의 관계는 형태적 거리가 아주 가까운 단어입니다.
  - 두 단어의 형태가 비슷하면서 한 단어의 빈도수가 매우 크다면 희귀한 단어를 오탈자로, 빈번한 단어를 정자로 생각할 수 있습니다.
  - 희귀한 단어를 빈번한 단어로 치환하여 오탈자를 수정합니다.
    - 써비스 → 서비스

Levenshtein (Edit) distance

- 가장 대표적인 string distance metric 입니다. 단어 A 에서 B 로 수정하기 위한 횟수를 거리로 정의합니다.
- 단어의 수정 방법은 세 가지로 정의됩니다.
  - **Deletion**: '서**어**비스' → '서비스' (거리 = 1)
  - **Insertion**: '데이터마닝' → '데이터마**이**닝' (거리 = 1)
  - **Substitution**: '데이**타**마이닝' → '데이**터**마이닝' (거리 = 1)

Levenshtein (Edit) distance

- String metric 중 가장 대표적인 척도입니다. 단어 A 에서 B 로 수정하기 위한 횟수를 거리로 정의합니다.

  - 데이타마닝 → 데이터마이닝

    1단계: 데이**타**마닝 → 데이**터**마닝 (누적 거리 = 1)

    2단계: 데이터마닝 → 데이터마**이**닝 (누적 거리 = 2)

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.
  - d(데이타마닝 → 데이터마이닝) = d(데이터마이닝 → 데이타마닝)
  - '데' → '데이타마닝'거리

	데	이	타	마	닝
데	0	1	2	3	4
이					
터					
마					
이					
닝					

Insertion [이, 타, 마, 닝]

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.
  - '데이' → '데이타마닝' 거리

	데	이	타	마	닝
데	0	1	2	3	4
이	1	0	1	2	3
터					
마					
이					
닝					

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.
  - '데이터' → '데이타마닝' 거리

	데	이	타	마	닝
데	0	1	2	3	4
이	1	0	1	2	3
터	2	1	1	2	3
마					
이					
닝					

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.
  - 첫 행과 열을 각각 [0, 1, 2,… m], [0, 1, 2, …, n] 로 초기화 합니다.

	데	이	타	마	닝
데	0	1	2	3	4
이	1				
터	2				
마	3				
이	4				
닝	5				

```
for j from 1 to (n=6):
    for i from 1 to (m=5):
        if s[i] = t[j]: substitutionCost := 0
        else: substitutionCost := 1
        d[i, j] := minimum(d[i-1, j] + 1, // deletion
                           d[i, j-1] + 1, // insertion
                           d[i-1, j-1] + substitutionCost) // subs.
return d[m, n]
```


* https://en.wikipedia.org/wiki/Levenshtein_distance

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.

길이가 1 짧은 단어로 바꾸는 것이 최소비용

	데	이	타	마	닝
데	0	1	2	3	4
이	1	0	1	2	3
터	2	1	1	2	3
마	3	2	2	1	2
이	4	3	3	2	2
닝	5	4	3	2	2

```
for j from 1 to (m=5):
    for I from 1 to (n=6):
        if s[i] = t[j]: substitutionCost := 0
        else: substitutionCost := 1
        d[i, j] := minimum(d[i-1, j] + 1, // deletion
                           d[i, j-1] + 1, // insertion
                           d[i-1, j-1] + substitutionCost) // subs.
return d[m, n]
```

d[3,2] = min(**d[2,2] + 1**,
d[3,1] + 1,
d[2,1] + subs)

* https://en.wikipedia.org/wiki/Levenshtein_distance

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.

'데이' 다음에 '타' 를 입력하는 것이 최소비용

	데	이	타	마	닝
데	0	1	2	3	4
이	1	0	1	2	3
터	2	1	1	2	3
마	3	2	2	1	2
이	4	3	3	2	2
닝	5	4	3	2	2

```
for j from 1 to (m=5):
    for i from 1 to (n=6):
        if s[i] = t[j]: substitutionCost := 0
        else: substitutionCost := 1
        d[i, j] := minimum(d[i-1, j] + 1, // deletion
                           d[i, j-1] + 1, // insertion
                           d[i-1, j-1] + substitutionCost) // subs.
return d[m, n]
```

d[2,3] = min(d[1,3] + 1,
d[2,2] + 1,
d[1,2] + subs)

* https://en.wikipedia.org/wiki/Levenshtein_distance

Levenshtein (Edit) distance

- 거리 행렬을 이용하여 edit distance 를 계산합니다.

'터' 대신 '타' 를 쓰는 것이 최소 비용

	데	이	타	마	닝
데	0	1	2	3	4
이	1	0	1	2	3
터	2	1	1	2	3
마	3	2	2	1	2
이	4	3	3	2	2
닝	5	4	3	2	2

```
for j from 1 to (m=5):
    for i from 1 to (n=6):
        if s[i] = t[j]: substitutionCost := 0
        else: substitutionCost := 1
        d[i, j] := minimum(d[i-1, j] + 1, // deletion
                           d[i, j-1] + 1, // insertion
                           d[i-1, j-1] + substitutionCost) // subs.
return d[m, n]
```

d[3,3] = min(d[2,3] + 1,
d[3,2] + 1,
d[2,2] + subs)

* https://en.wikipedia.org/wiki/Levenshtein_distance

Jaccard distance

- Jaccard distance 는 집합 간의 유사도를 정의하는 방법입니다.
- 각 단어를 글자 (units) 의 집합으로 생각합니다.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard distance

- 각 단어를 글자 (units) 의 집합으로 생각합니다.

$$J('데이터마이닝', '데이타마닝') = \frac{|\{데, 이, 마, 닝\}|}{|\{데, 이, 터, 타, 마, 닝\}|} = \frac{\mathbf{4}}{\mathbf{6}}$$

- Similarity 를 distance 로 변환합니다. $J_D(A, B) = 1 - J(A, B)$
- **한 글자(unit)**의 등장 횟수는 고려하지 않습니다.

Cosine distance

- Cosine 은 단어 별로 글자 (units) 의 등장 횟수를 고려합니다.
  - 글자 (혹은 units) 가 features 로 정의된 벡터입니다.

	데	이	터	타	마	닝
데이터마이닝	1	2	1	0	1	1
데이타마닝	1	1	0	1	1	1

  - $\cos_D(A, B) = 1 - \cos(A, B)$
  - $\cos_D$(데이터마이닝, 데이터마닝) = 0.79

한국어를 위한 String distance

- String distance 의 unit 을 '초/중/종성' 이나 '어절' 로 정의할 수 있습니다.

  - d(서비스, **써**비스) != d(서비스, **자**비스)
  - Unit = 초/중/종성
    - d(서비스, 써비스) = **1 → 0.33**
  - Unit = 어절
    - d('어디야 지금', '지금 어디야') = **6 → 2**

한국어를 위한 String distance

- Character n-gram 을 units 으로 이용할 수 있습니다.

$$J('데이터마이닝','데이타마닝') = \frac{|\{데이\}|}{|\{데이, 이터, 터마, 마이, 이닝, 이타, 타마, 마닝\}|} = \frac{\mathbf{1}}{\mathbf{8}}$$

한국어를 위한 String distance

- n 음절을 unit 으로 이용하면 더 정확한 문맥을 반영할 수 있습니다.

$$J('자살금지','지금살자') = 1 - \frac{|\{자, 살, 금, 지\}|}{|\{자, 살, 금, 지\}|} = 0$$

$$J('자살금지','지금살자') = 1 - \frac{|\{\ \}|}{|\{자살, 살금, 금지, 지금, 금살, 살자\}|} = 1$$

한국어를 위한 String distance

- Edit distance 에서 (a → b) 마다 다른 비용을 부여할 수 있습니다.
  - d(서비스, **써**비스) < d(서비스, **자**비스)
  - cost(서, 써) < cost(서, 자)

```
for j from 1 to (n=6):
    for i from 1 to (m=5):
        if s[i] = t[j]: substitutionCost := 0
        else: substitutionCost := cost.get(s[i], t[j])

        d[i, j] := minimum(d[i-1, j] + 1, // deletion
                           d[i, j-1] + 1, // insertion
                           d[i-1, j-1] + substitutionCost) // subs.
return d[m, n]
```

한국어를 위한 String distance

- 한글은 초/중/종성 단위로 Edit distance 를 적용할 수 있습니다.
  - 음절 단위의 edit distance 에서는 d('가', '감') = 1 입니다.
  - 초/중/종성 단위의 edit distance 에서는 d('가', '감') = 1/3 입니다.
    - 혹은 초/중/종성별로 다르게 가중치를 부여할 수도 있습니다.

한국어를 위한 String distance

- 유니코드에서 한글과 자음/모음은 특정한 범위를 지닙니다.

글자 범위	유니코드 범위
가 ~ 힣 (한글)	44032 ~ 55203
ㄱ ~ ㅎ (자음)	12593 ~ 12622
ㅏ ~ ㅣ (모음)	12623 ~ 12643
a ~ z (알파벳 소문자)	97 ~ 122
A ~ Z (알파벳 대문자)	65 ~ 90
0 ~ 9 (숫자)	48 ~ 57

한국어를 위한 String distance

- 유니코드에서 한글과 자음/모음은 특정한 범위를 지닙니다.
- Python 의 ord() 함수를 이용하면 글자의 유니코드 값을 얻을 수 있습니다.

```
for char in 'azAZ가힣ㄱㄴㅎㅏ':
    print('{} == {}'.format(char, ord(char)))
```

```
a == 97
z == 122
A == 65
Z == 90
가 == 44032
힣 == 55203
ㄱ == 12593
ㄴ == 12596
ㅎ == 12622
ㅏ == 12623
```

한국어를 위한 String distance

- 유니코드에는 한글의 초/중/종성의 결합 규칙이 있습니다

```
def decompose(c):
    if not character_is_korean(c):
        return None
    i = ord(c)
    if (jaum_begin <= i <= jaum_end):
        return (c, ' ', ' ')
    if (moum_begin <= i <= moum_end):
        return (' ', c, ' ')
    # decomposition rule
    i -= kor_begin
    cho = i // chosung_base
    jung = ( i - cho * chosung_base ) // jungsung_base
    jong = ( i - cho * chosung_base - jung * jungsung_base )
    return (chosung_list[cho], jungsung_list[jung], jongsung_list[jong])
```

```
chosung_list = [
  'ㄱ', 'ㄲ', 'ㄴ', 'ㄷ', 'ㄸ',
  'ㄹ', 'ㅁ', 'ㅂ', 'ㅃ', 'ㅅ',
  'ㅆ', 'ㅇ' , 'ㅈ', 'ㅉ', 'ㅊ',
  'ㅋ', 'ㅌ', 'ㅍ', 'ㅎ']
jungsung_list = [
  'ㅏ', 'ㅐ', 'ㅑ', 'ㅒ', 'ㅓ',
  'ㅔ', 'ㅕ', 'ㅖ', 'ㅗ', 'ㅘ',
  'ㅙ', 'ㅚ', 'ㅛ', 'ㅜ', 'ㅝ',
  'ㅞ', 'ㅟ', 'ㅠ', 'ㅡ', 'ㅢ', 'ㅣ']
jongsung_list = [
  ' ', 'ㄱ', 'ㄲ', 'ㄳ', 'ㄴ',
  'ㄵ', 'ㄶ', 'ㄷ', 'ㄹ', 'ㄺ',
  'ㄻ', 'ㄼ', 'ㄽ', 'ㄾ', 'ㄿ',
  'ㅀ', 'ㅁ', 'ㅂ', 'ㅄ', 'ㅅ',
  'ㅆ', 'ㅇ', 'ㅈ', 'ㅊ', 'ㅋ',
  'ㅌ', 'ㅍ', 'ㅎ']
```

한국어를 위한 String distance

- 유니코드에는 한글의 초/중/종성의 결합 규칙이 있습니다

```
def compose(chosung, jungsung, jongsung):
    char = (
        kor_begin +
        chosung_base * chosung_list.index(chosung) +
        jungsung_base * jungsung_list.index(jungsung) +
        jongsung_list.index(jongsung)
    )
    return char
```

한국어를 위한 String distance

- Edit distance 의 substitution cost 를 아래와 같은 함수로 정의합니다.
  - 두 글자가 같을 경우의 비용은 0 입니다.
  - 두 글자가 다르다면 초/중/종성을 분해한 길이가 3 인 list of str 에 대한 edit distance 를 계산합니다.
  - 정규화 (normalize) 를 위하여 그 값을 3으로 나눠줍니다.

```
def substitution_cost(c1, c2):
    if c1 == c2:
        return 0
    return levenshtein(decompose(c1), decompose(c2))/3
```

Inverted index 를 이용한 빠른 Edit distance

- Edit distance 는 각 글자를 unit 으로 이용합니다.
- Edit distance 가 작은 두 단어는 길이가 비슷하고 겹치는 단어가 많아야 합니다.
  - 단어는 bag of characters 로 표현할 수 있습니다.
  - Sparse vector 이기 때문에 inverted index 를 이용할 수 있습니다.

Inverted index 를 이용한 빠른 Edit distance

- Edit distance 의 길이가 d 이하인 두 단어는 다음의 조건을 만족합니다.
  - $n_1 = len(w_1), n_2 = len(w_2)$ 일 때, $|n_1 - n_2| \leq d$
    - 두 단어의 길이의 차이는 d 보다 작다.
  - $len\big(set(w_1)\big) - len\big(set(w_1) \cap set(w_2)\big) \leq d$
    - 두 단어에 공통으로 있지 않는 글자의 숫자의 개수가 d 보다 작다.

Inverted index 를 이용한 빠른 Edit distance

- Character 기준으로 inverted index 를 만듭니다.
  - Index[c] = [word1, word2, …]
- 주어진 query word 의 characters 를 이용하여 다음의 조건을 만족하는 후보를 탐색합니다.
  - 두 단어의 길이의 차이는 d 보다 작다.
  - 두 단어에 공통으로 있지 않는 글자의 숫자의 개수가 d 보다 작다.

Inverted index 를 이용한 빠른 Edit distance

- 132,864 개의 단어 사전에서 Edit distance 를 이용하여 거리가 1 이하인 단어를 탐색하는 예시입니다.
  - 132,864 개의 단어와의 거리계산 및 정렬을 위하여 약 2.49 초가 소요됩니다.
  - Inverted index 를 이용하면 7 개의 단어와 실제 거리 계산을 수행합니다.
    - 0.0056 초가 소요됩니다.

https://github.com/lovit/inverted_index_for_hangle_editdistance

Inverted index 를 이용한 빠른 Edit distance

```
from fast_hangle_levenshtein import LevenshteinIndex


nouns = ['양식어가', '식품유통사', 'ETN전담팀', '도로주행', '로우프라이스펀드', …]
print(len(nouns)) # 132,864
indexer = LevenshteinIndex(nouns, verbose=True)
print(indexer.levenshtein_search('분식회계'))
```

```
[('분식회계', 0), ('분식회', 1), ('분식회계설', 1), ('분석회계', 1)]
```


https://github.com/lovit/inverted_index_for_hangle_editdistance