문서 거리 문제

1. 문서 거리 문제란?

* 두 문서 D1과 D2 사이의 거리를 계산하는 것이다.

1. 계산하는 이유?

* 문서 간의 유사성을 탐색할 수 있다.

Ex) 유사 페이지 탐색, 과제 부정행위 탐색 등

1. 두 문서의 유사성을 어떻게 알 수 있을까?

* 두 문서를 벡터라고 하자.
* 두 문서는 각각 길이를 가지고 있는데, 이 길이가 바로 각 문서가 가지고 있는 단어의 개수가 된다.
* 그리고 두 문서의 차이는, 두 벡터의 차이이므로 두 벡터 사이인 사잇각을 구해주면 두 문서의 차이를 알 수 있다.
* 그리고 두 문서에 들어있는 단어와 단어의 개수를 정리하여 점곱했을 때 점수가 높을 수록 공통된 단어들이 많다는 것을 알 수 있다.

1. 사잇각을 구하는 방법?

* acos(두 벡터의 점곱) / 두 벡터의 norm의 곱
* 위와 같이 하면 라디안 값으로 사잇각을 구할 수 있다.
* Degree로 구하려면 위의 값에 degrees함수를 사용해주면 된다.

1. 연산 과정
2. 문서를 단어들로 쪼갠다.
3. 단어의 빈도를 계산하여 저장한다
4. 벡터의 점곱을 구한다.
5. 점곱을 빈도로 나눠준 후 그 값을 역코사인 한다.

강의에서 제공해준 코드는 총 8개로 코드 파일이 변할 때마다 비용이 절감되도록 했다.

그리하여 결과적으로 1번째 파일과 8번째 파일의 비용은 1mb파일을 기준으로 약 228초의 차이가 나타났다.

* 개선과정
* 1. 문자열을 + 연산자로 병합하는 것이 아닌 extend연산자로 병합
* 2. 점곱의 방식을 변화하여 단어의 빈도 계산
* 기존에는 이중반복문을 사용하여 단어가 같을 때 점곱을 해줬다.
* 변화한 후

(다음 페이지)

* sum = 0.0
* i = 0
* j = 0
* while i<len(L1) and j<len(L2):
* # L1[i:] and L2[j:] yet to be processed
* if L1[i][0] == L2[j][0]:
* # both vectors have this word
* sum += L1[i][1] \* L2[j][1]
* i += 1
* j += 1
* elif L1[i][0] < L2[j][0]:
* # word L1[i][0] is in L1 but not L2
* i += 1
* else:
* # word L2[j][0] is in L2 but not L1
* j += 1
* return sum

3. 단어의 개수를 세는 방식에 변화를 줬다.

-> 기존에는 문서를 돌면서 새로운 단어가 나올 때마다 list 안에 [new\_word, 1]을 append 해줬다. 그리고 선형 탐색으로 원래 있는 단어로 탐색됐을 시, word[i] + 1로 그 빈도를 더해줬다.

-> 변화 후에는 dict를 사용해서 단어와 그 개수를 저장한다.

4. 문서의 단어를 쪼개는 방법에 변화를 줬다.

-> 기존에는 문자 하나하나를 알파벳 혹은 숫자가 맞는지 체크해준 후에 그 문자들을 리스트에 담아서 소문자로 변환해준 후 리스트에 담는 것을 반복했다.

-> 변화한 후에는 python의 translate를 사용하여 translate 함수 처리를 해준 line을 word\_list에 담는다.

translation\_table = string.maketrans(string.punctuation+string.uppercase,

" "\*len(string.punctuation)+string.lowercase)

def get\_words\_from\_string(line):

"""

Return a list of the words in the given input string,

converting each word to lower-case.

Input: line (a string)

Output: a list of strings

(each string is a sequence of alphanumeric characters)

"""

line = line.translate(translation\_table)

word\_list = line.split()

return word\_list

5. 문서의 단어를 알파벳순으로 정렬하는 방식에 변화를 줬다.

-> 기존에는 삽입정렬을 사용했다.

-> 변화한 후에는 합병정렬을 사용했다.

6. 점곱을 하는 방식에 변화를 줬다.

-> 2단계에서 방법을 한번 바꿨으나, 이번에는 빈도 계산을 위해 만들었던 dict를 활용하여 점곱을 하는 방식으로 변화를 줬다.

def inner\_product(D1,D2):

"""

Inner product between two vectors, where vectors

are represented as dictionaries of (word,freq) pairs.

Example: inner\_product({"and":3,"of":2,"the":5},

{"and":4,"in":1,"of":1,"this":2}) = 14.0

"""

sum = 0.0

for key in D1:

if key in D2:

sum += D1[key] \* D2[key]

return sum

7. 문서에서 가져온 글자들을 라인 단위로 변환을 해주는 것이 아닌 문서 단위로 쪼개도록 변환한다.

-> 기존에는 한 줄 씩 변환한 문자들을 리스트에 담았다.

-> 변화한 후에는 텍스트 전체를 word\_list에 담도록 한다.

⭕️ 더 나은 개선을 위해 시도해본 것

1. 정규표현식 사용

->

def get\_words\_from\_line\_list(text, filename):

"""

Parse the given text into words.

Return list of all words found.

"""

# text = text.translate(translation\_table)

text = re.findall(r'\w+', filename)

word\_list = text.split()

return word\_list

문서를 담을 때, 정규표현식을 사용해서 한번에 담도록 설정해보았다.

결과는 다음 페이지

테이블이(가) 표시된 사진

자동 생성된 설명

정규표현식을 사용하지 않고 문서 단위로 list에 단어들을 translate하여 담았을 때와 같게 나왔다.

하지만, 이 결과는 정규표현식이 좋은 방안이라는 것을 말해주는 건 아니라고 생각한다.

혹시 몰라서 찾아보니 정규표현식은 최대 O(2^n) 즉, 지수 시간까지 비용을 잡아먹는 모듈이라고 한다.

내가 정확하게 실험을 한 것인지는 모르겠지만 컴퓨터 성능의 발전으로 인해 전체적으로 연산이 빨라져서 비용이 적게 들게 된 점이 있지 않나 싶다.

Docdist1.py도 200초대가 아닌 docdist.8.py와 거의 비슷하게 비용이 나왔기 때문이다.