2018.10.04

자연어 처리

3기 이주영

Growth Hackers

CONTENTS

1. NLP
2. 텍스트 전처리 과정
3. KoNLPy / nltk 실습
4. LDA
5. Word2Vec

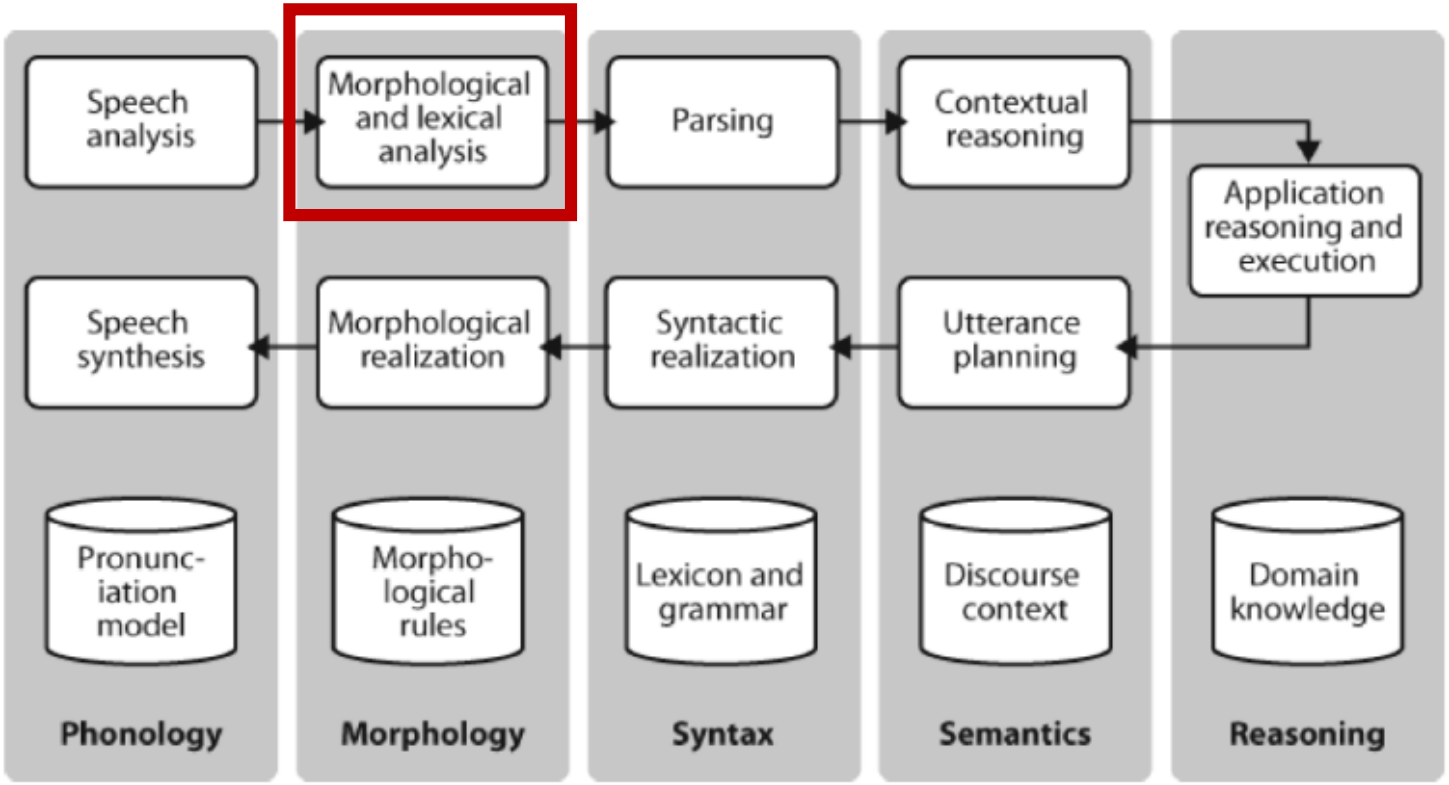
1. NLP

- Natural Language Processing(자연어 처리)
  - 텍스트에서 의미 있는 정보를 분석, 추출하고 이해하는 일련의 기술 집합
  - 기계어(0,1)과 인간 언어의 소통. 컴공 + 인공지능 + 전산언어학

- 인간이 발화하는 언어 현상을 기계적으로 분석해서 컴퓨터가 이해할 수 있는 형태로 만드는 자연 언어 이해 혹은 그러한 형태를 다시 인간이 이해할 수 있는 언어로 표현하는 제반 기술

2. 텍스트 전처리 과정

- 자연언어처리의 기본 절차



음성인식, 형태소 분석, 파싱 (문장의 문법적 구조 분석) 등이 각각 언어학의 음운/형태/통사론 등에 대응된다!

Growth Hackers

2. 텍스트 전처리 과정

- 어휘 분석(Lexical Analysis) 절차

1. 문장 분리(Sentence splitting)
   - Corpus를 문장 단위로 끊어서 입력
   - 일반적으로는 마침표(.), 느낌표(!), 물음표(?) 등을 기준으로 수행

2. Tokenize
   - token(토큰)이란 의미를 가지는 문자열, 형태소나 단어까지 포함
   - tokenizing이란 문서나 문장을 분석하기 좋도록 토큰으로 나누는 작업

2. 텍스트 전처리 과정

- 어휘 분석(Lexical Analysis) 절차

3. Morphological analysis(형태소 분석)
   - Text Normalization이라고도 함
   - 토큰들을 좀 더 일반적인 형태로 분석해 단어 수를 줄여 분석의 효율성을 높임
   - stemming(단어를 축약형으로 변환) / lemmatization(기본형으로 변환)

4. Part-Of-Speech(POS) Tagging
   - 토큰의 품사 정보를 할당하는 작업

*KoNLPy는 문장 분리, tokenize, lemmatization, pos tagging에 이르는 전 과정을 한꺼번에 수행해 줌

3. KoNLPy / NLTK 실습

- **KoNLPy**
  - 설치완료!
  - 한국어 텍스트를 이용하여 기초적인 NLP 작업 가능
  - Kkma, Twitter, Komoran, Mecab 등

- 빠른 분석이 중요할 때 : **트위터**
- 정확한 품사 정보가 필요할 때 : **꼬꼬마**
- 정확성, 시간 모두 중요할 때 : **코모란**

- **NLTK**
  - 설치하기 : cmd => pip install nltk
  - 영어 텍스트를 이용하여 기초적인 NLP 작업 가능

3. KoNLPy / NLTK 실습

- Read document(NLTK, KoNLPy에서 제공되는 문서 사용)

```
import nltk
import konlpy
#nltk 데이터 다운로드
nltk.download('gutenberg')
nltk.download('maxent_treebank_pos_tagger')
```

```
from nltk.corpus import gutenberg    # Docs from project gutenberg.org
files_en = gutenberg.fileids()       # Get file ids
doc_en = gutenberg.open('shakespeare-hamlet.txt').read()

from konlpy.corpus import kobill     # Docs from pokr.kr/bill
files_ko = kobill.fileids()          # Get file ids
doc_ko = kobill.open('1809890.txt').read()
```

3. KoNLPy / NLTK 실습

- Tokenize

```
tokens_en = nltk.word_tokenize(doc_en)
print(tokens_en)
```

```
['[', 'The', 'Tragedie', 'of', 'Hamlet', 'by', 'William', 'Shakespeare', '15
99', ']', 'Actus', 'Primus', '.', 'Scoena', 'Prima', '.', 'Enter', 'Barnard
o', 'and', 'Francisco', 'two', 'Centinels', '.', 'Barnardo', '.', 'Who',
"'s", 'there', '?', 'Fran', '.', 'Nay', 'answer', 'me', ':', 'Stand', '&',
'vnfold', 'your', 'selfe', 'Bar', '.', 'Long', 'liue', 'the', 'King', 'Fra
```

```
from konlpy.tag import Kkma;kkma = Kkma()
tokens_ko = kkma.morphs(doc_ko)
print(tokens_ko)
```

```
['지방', '공무원', '법', '일부', '개정', '법률안', '(', '정의', '화', '의
원', '대표', '발의', ')', '의', '안', '벌', 'ㄴ', '호', '9890', '발', '의',
'연월일', ':', '2010', '.', '11', '.', '12', '.', '발', '의', '자', ':', '정
의', '화', '.', '이명수', '.', '김', '을', '동', '이사철', '.', '여', '상규',
```

3. KoNLPy / NLTK 실습

- Load tokens with `nltk.Text()`

```
en = nltk.Text(tokens_en)
```

```
ko = nltk.Text(tokens_ko, name='대한민국 국회 의안 제 1809890호')
```

```
print(len(en.tokens))       # returns number of tokens (document length)
print(len(set(en.tokens)))  # returns number of unique tokens
en.vocab()                  # returns frequency distribution
```

```
36326
5540

FreqDist({',': 2892, '.': 1879, 'the': 860, 'and': 605, 'of': 576, 'to': 574,
':': 566, 'I': 550, 'you': 474, '?': 459, ...})
```

```
print(len(ko.tokens))       # returns number of tokens (document length)
print(len(set(ko.tokens)))  # returns number of unique tokens
ko.vocab()                  # returns frequency distribution
```

```
1777
476

FreqDist({'₩n₩n': 127, '.': 49, '의': 46, '육아휴직': 38, '을': 28, '(': 27,
')': 26, '이': 25, '자': 24, '에': 23, ...})
```

3. KoNLPy / NLTK 실습

- `nltk.Text()` 가 제공하는 다양한 기능

```
ko.concordance('초등학교')
```

'초등학교'라는 단어가 포함된 부분 찾아주기

```
ko.count('초등학교')
```

'초등학교'라는 단어가 등장한 횟수 찾아주기

```
ko.similar('자녀')
ko.similar('육아휴직')
```

해당 단어와 유사한 단어 찾아주기

3. KoNLPy / NLTK 실습

- POS tagging

```
#nltk.download('averaged_perceptron_tagger')
tokens = "The little yellow dog barked at the Persian cat".split()
tags_en = nltk.pos_tag(tokens)
tags_en
```

```
[('The', 'DT'),
 ('little', 'JJ'),
 ('yellow', 'JJ'),
 ('dog', 'NN'),
 ('barked', 'VBD'),
 ('at', 'IN'),
 ('the', 'DT'),
 ('Persian', 'JJ'),
 ('cat', 'NN')]
```

3. KoNLPy / NLTK 실습

- POS tagging

```
from konlpy.tag import Kkma; kkma = Kkma()
tags_ko = kkma.pos("작고 노란 강아지가 페르시안 고양이에게 짖었다")
tags_ko
```

```
[('작고', 'NNG'),
 ('노', 'NNG'),
 ('이', 'VCP'),
 ('란', 'ETD'),
 ('강아지', 'NNG'),
 ('가', 'JKS'),
 ('페르', 'NNG'),
 ('시안', 'NNP'),
 ('고양이', 'NNG'),
 ('에게', 'JKM'),
 ('짖', 'VV'),
 ('었', 'EPT'),
 ('다', 'EFN')]
```

4. LDA

- Latent Dirichlet allocation(잠재 디리클레 할당)

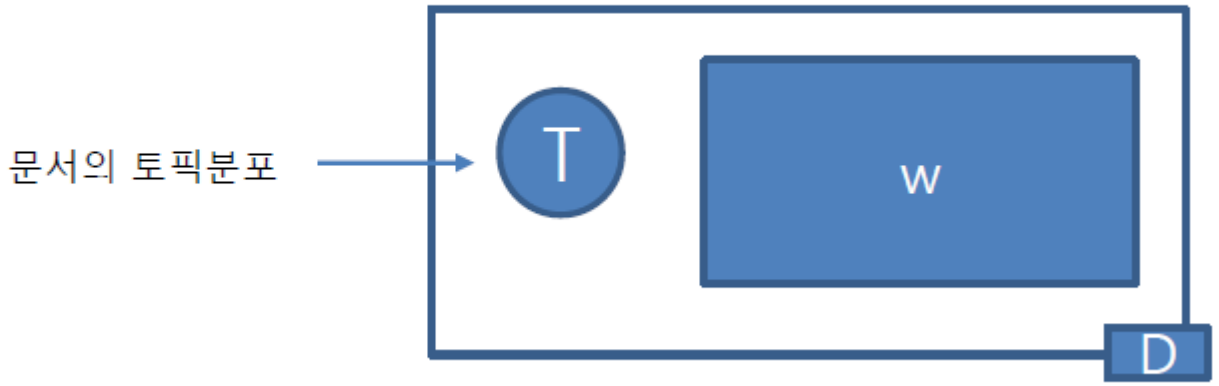
  - 주어진 문서에 대하여 각 문서에 어떤 주제들이 존재하는지에 대한 확률적 토픽 모델 기법 중 하나
  - 토픽 별 단어의 분포, 문서 별 토픽의 분포를 모두 추정

  - 문서의 생성 과정을 가정
  - 문서 내의 각 단어는 문서의 토픽 분포로부터 먼저 임의의 토픽이 선택된 뒤, 토픽의 단어 분포로부터 생성되었다고 가정

4. LDA

- 문서의 생성 과정(가정)

1. 문서의 토픽분포가 존재한다고 가정한다.

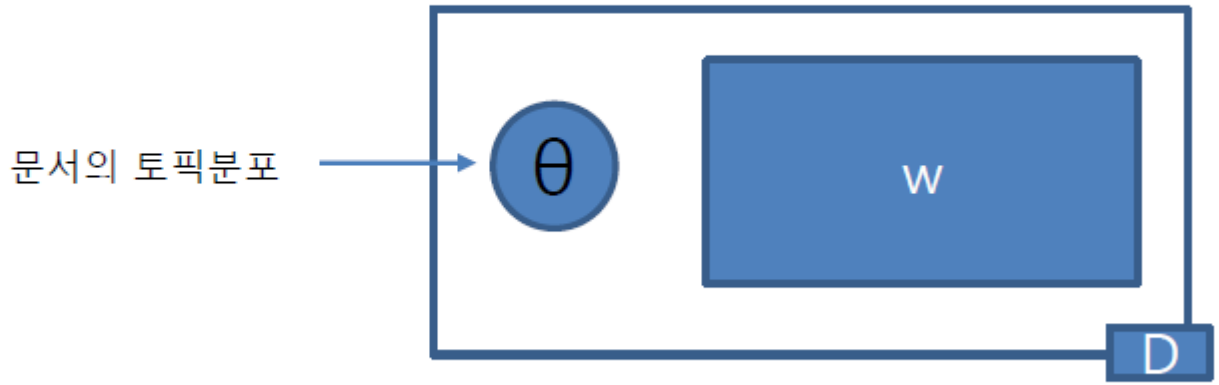


Growth Hackers

4. LDA

- 문서의 생성 과정(가정)

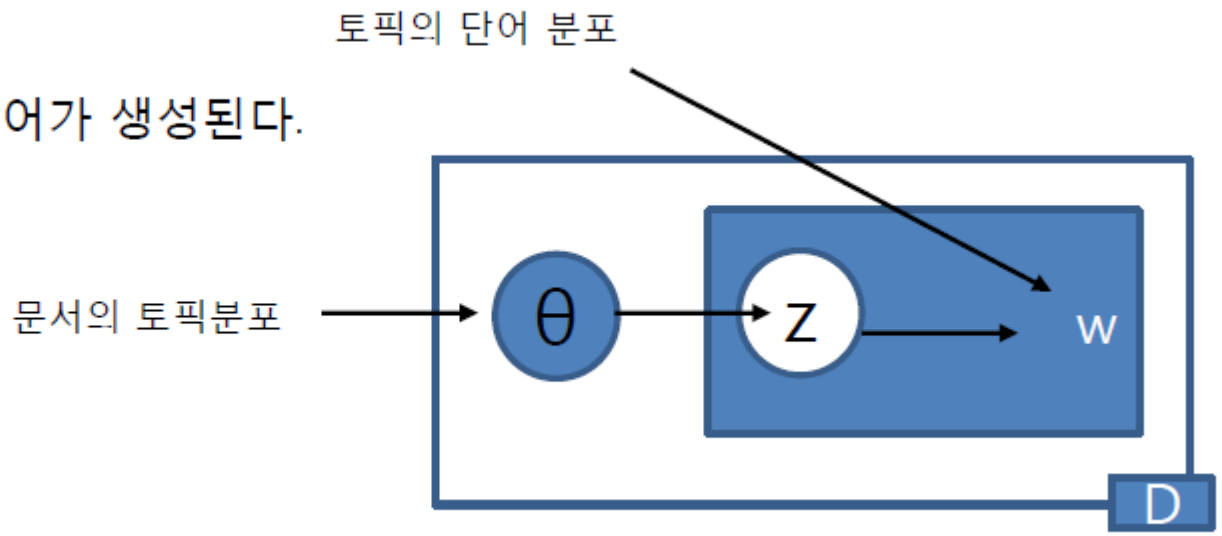
2. 문서의 토픽분포로부터 임의의 토픽(θ)를 선택한다.



4. LDA

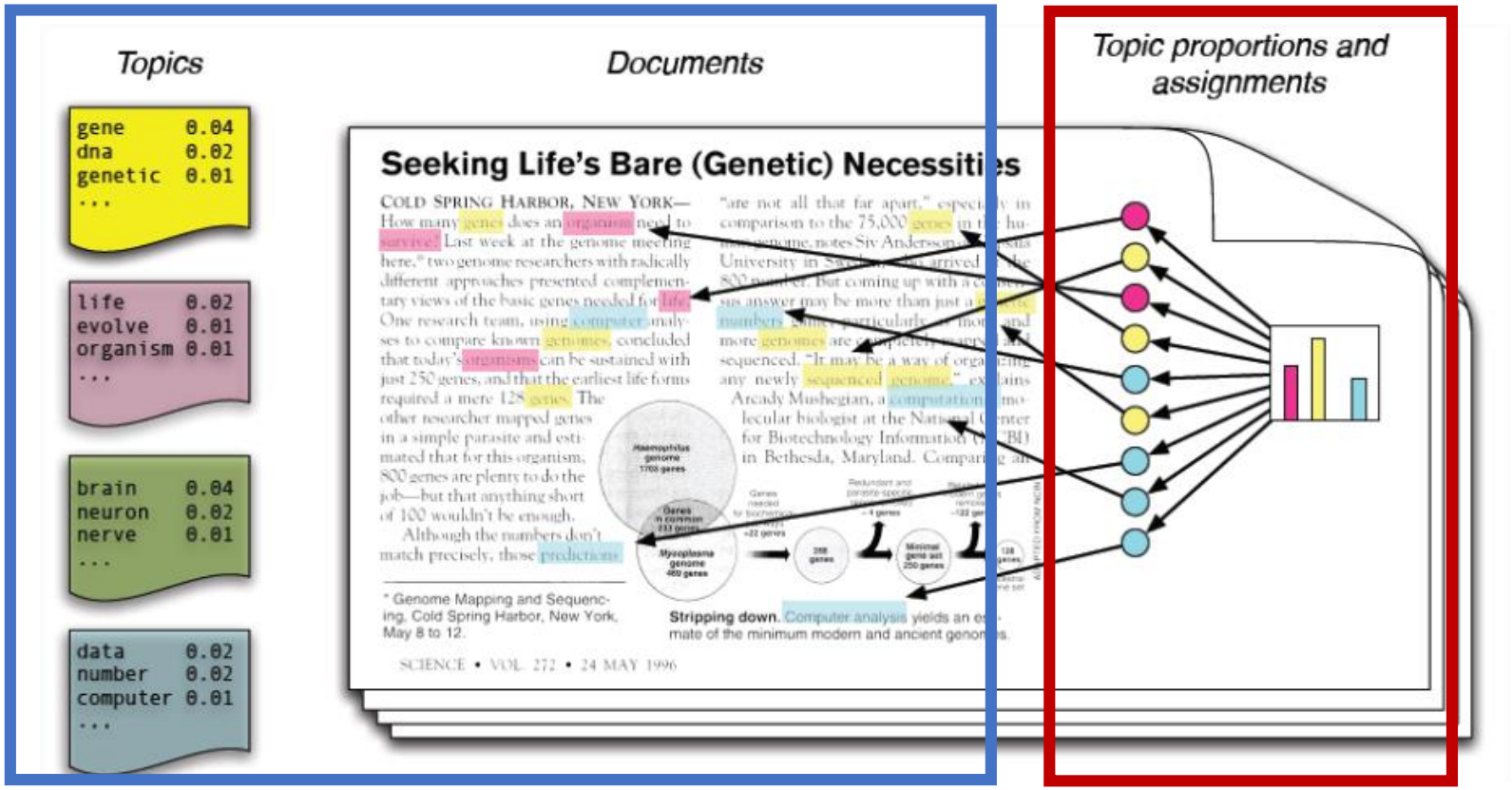
- 문서의 생성 과정(가정)

3. 토픽의 단어분포로부터 단어가 생성된다.



Growth Hackers

4. LDA



LDA의 산출 결과물

LDA 핵심 프로세스

문서가 생성되는 과정을 확률모형으로 모델링

단어의 잠재(Latent)정보를 알아내는 과정

Growth Hackers

4. LDA

- 수식
  - d번째 문서 i번째 단어의 토픽 $z_{d,i}$가 j번째에 할당될 확률

$$p(z_{d,i}=j|z_{-i},w) = \frac{n_{d,k}+\alpha_k}{\sum_{i=1}^{K}(n_{d,i}+\alpha_i)} \times \frac{v_{k,w_{d,n}}+\beta_{w_{d,n}}}{\sum_{j=1}^{V} v_{k,j}+\beta_j} = AB$$

표기	내용
$n_{d,k}$	k번째 토픽에 할당된 d번째 문서의 단어 빈도
$v_{k,w_{d,n}}$	전체 말뭉치에서 k번째 토픽에 할당된 단어 $w_{d,n}$의 빈도
$w_{d,n}$	d번째 문서에 n번째로 등장한 단어
α_k	문서의 토픽 분포 생성을 위한 디리클레 분포 파라메터
β_k	토픽의 단어 분포 생성을 위한 디리클레 분포 파라메터
K	사용자가 지정하는 토픽 수
V	말뭉치에 등장하는 전체 단어 수
A	d번째 문서가 k번째 토픽과 맺고 있는 연관성 정도
B	d번째 문서의 n번째 단어($w_{d,n}$)가 k번째 토픽과 맺고 있는 연관성 정도

4. LDA

- LDA 구현 - 데이터

```
#문서의 집합 documents
documents = [["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]]
```

4. LDA

- LDA 구현 – 변수 선언

```
#조건부 확률 분포 정의를 위한 준비

#1. 각 토픽이 각 문서에 할당되는 횟수
#counter로 구성된 list
#각각의 counter는 각 문서를 의미함
document_topic_counts = [Counter() for _ in documents]

#2. 각 단어가 각 토픽에 할당되는 횟수
# 각각의 counter는 각 토픽을 의미함
topic_word_counts = [Counter() for _ in range(K)]
```

document_topic_counts[2][0]
> 문서 3에서 토픽 1과 관련 있는 단어 수

topic_word_counts[0]['Java']
> 'Java'와 토픽 1이 연관 지어 등장한 횟수

4. LDA

- LDA 구현 – 변수 선언

```
#3. 각 토픽에 할당되는 총 단어 수
# 각각의 숫자는 각 토픽을 의미함
topic_counts = [0 for _ in range(K)]

#4. 각 문서에 포함되는 총 단어의 수
# 각각의 숫자는 각 문서를 의미함
document_lengths = [len(d) for d in documents]

#5. 단어 종류의 수
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)

#6. 총 문서의 수
D = len(documents)
```

```
{'statistics', 'scikit-learn', 'machine learning', 'R', 'pandas', 'C++', 'MongoDB',
 'deep learning', 'numpy', 'Python', 'decision trees', 'databases', 'MySQL',
'NoSQL', 'probability', 'Cassandra', 'neural networks', 'HBase', 'MapReduce',
'Postgres', 'programming languages', 'regression', 'Haskell', 'support vector
machines', 'mathematics', 'artificial intelligence', 'Hadoop', 'Storm',
'libsvm', 'Mahout', 'Java', 'scipy', 'Spark', 'statsmodels', 'Big Data', 'theory'}
```

4. LDA

- LDA 구현 – 새로운 topic 계산하기

  - d번째 문서 i번째 단어의 토픽 $z_{d,i}$가 j번째에 할당될 확률은 AB

```
def p_topic_given_document(topic, d, alpha=0.1):
    # 문서 d의 모든 단어 가운데 topic에 속하는
    # 단어의 비율 (alpha를 더해 smoothing)
    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    # topic에 속한 단어 가운데 word의 비율
    # (beta를 더해 smoothing)
    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + V * beta))

def topic_weight(d, word, k):
    # 문서와 문서의 단어가 주어지면
    # k번째 토픽의 weight를 반환
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)
```

A = p_topic_given_document

B = p_word_given_topic

AB = topic_weight

Growth Hackers

4. LDA

- LDA 구현 – 새로운 topic 계산하기
  - AB를 바탕으로 샘플링을 하여 $z_{d,i}$에 새로운 topic 할당

```
def choose_new_topic(d, word):
    return sample_from([topic_weight(d, word, k) for k in range(K)])

#랜덤으로 생성된 weight로부터 인덱스를 생성함
def sample_from(weights):
     total = sum(weights)
     rnd = total * random.random()       # uniform between 0 and total
     for i, w in enumerate(weights):
         rnd -= w                        # return the smallest i such that
         if rnd <= 0: return i           # sum(weights[:(i+1)]) >= rnd
```

4. LDA

- Document_topic을 생성하는 깁스 샘플링

1. 모든 문서의 모든 단어에 임의의 토픽을 부여
2. 토픽-단어 분포 & 문서-토픽 분포 ~> 각 토픽에 weight 할당
3. Weight를 사용하여, 해당 단어에 알맞은 새로운 토픽 할당
4. 이 과정의 반복
5. 토픽-단어 분포 & 문서-토픽 분포의 결합확률로부터 나오는 표본 획득

4. LDA

- LDA 구현 – 깁스 샘플링

```
random.seed(0)

#topic의 개수
K = 4

# 각 단어를 임의의 토픽에 배정
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

# 랜덤 초기화한 상태에서 AB를 구하는 데 필요한 숫자 계산하기
for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

```
# 조건부 확률 분포를 이용하여 (토픽-단어), (문서-토픽)에 대한 깁스 샘플링 실행하기
for iter in range(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):


            # remove this word / topic from the counts
            # so that it doesn't influence the weights
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # choose a new topic based on the weights
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic


            # and now add it back to the counts
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

4. LDA

- LDA 구현 – 결과

```
#각 토픽에 가장 영향력이 높은 (weight)값이 큰 단어 탐색
for k, word_counts in enumerate(topic_word_counts):
        for word, count in word_counts.most_common():
            if count > 0: print (k, word, count)
```

```
0 Java 3
0 Big Data 3
0 Hadoop 2
0 HBase 1
0 C++ 1
0 Spark 1
0 Storm 1
0 programming languages 1
0 MapReduce 1
0 Cassandra 1
0 deep learning 1
1 HBase 2
1 neural networks 2
1 Postgres 2
1 MongoDB 2
1 machine learning 2
1 Cassandra 1
1 numpy 1
1 decision trees 1
1 deep learning 1
1 databases 1
1 MySQL 1
1 NoSQL 1
1 artificial intelligence 1
1 scipy 1
2 regression 3
2 Python 2
2 R 2
2 libsvm 2
2 scikit-learn 2
2 mathematics 1
2 support vector machines 1
2 Haskell 1
2 Mahout 1
3 statistics 3
3 probability 3
3 Python 2
3 R 2
3 pandas 2
3 statsmodels 2
3 C++ 1
3 artificial intelligence 1
3 theory 1
```

4. LDA

- LDA 구현 – 결과

```
topic_names = ["Big data and programming languages",
                   "python and statistics",
                   "databases",
                   "machine learning"]

for document, topic_counts in zip(documents, document_topic_counts):
        print (document)
        for topic, count in topic_counts.most_common():
            if count > 0:
                print (topic_names[topic], count)
```

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big data and programming languages 7
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
python and statistics 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
python and statistics 2
databases 2
machine learning 2
['R', 'Python', 'statistics', 'regression', 'probability']
machine learning 3
databases 2
['machine learning', 'regression', 'decision trees', 'libsvm']
databases 2
python and statistics 2
['Python', 'R', 'Java', 'C++', 'Haskell', 'programming languages']
databases 3
Big data and programming languages 3
['statistics', 'probability', 'mathematics', 'theory']
machine learning 3
databases 1
['machine learning', 'scikit-learn', 'Mahout', 'neural networks']
databases 2
python and statistics 2
['neural networks', 'deep learning', 'Big Data', 'artificial intelligence']
python and statistics 3
Big data and programming languages 1
['Hadoop', 'Java', 'MapReduce', 'Big Data']
Big data and programming languages 4
['statistics', 'R', 'statsmodels']
machine learning 3
['C++', 'deep learning', 'artificial intelligence', 'probability']
machine learning 3
Big data and programming languages 1
['pandas', 'R', 'Python']
machine learning 3
['databases', 'HBase', 'Postgres', 'MySQL', 'MongoDB']
python and statistics 5
['libsvm', 'regression', 'support vector machines']
databases 3
```

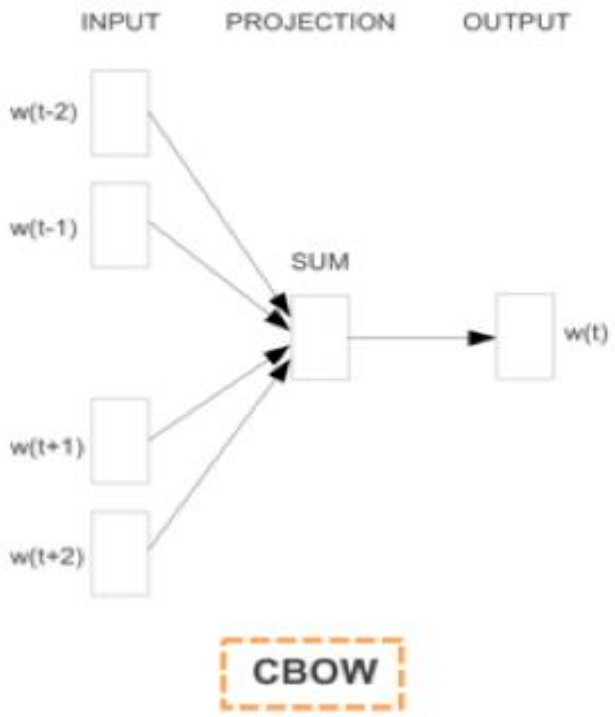
5. Word2Vec

- Word2Vec
  - 단어를 벡터화하는 임베딩(embedding) 방법론
  - 텍스트를 처리하는 인공 신경망

  - **CBOW**(Continuous Bag of Words)와 **Skip-Gram** 두 가지 방식
    : 전자는 주변에 있는 단어들을 가지고 중심에 있는 단어를 맞추는 방식이고, 후자는 중심에 있는 단어로 주변 단어를 예측하는 방법

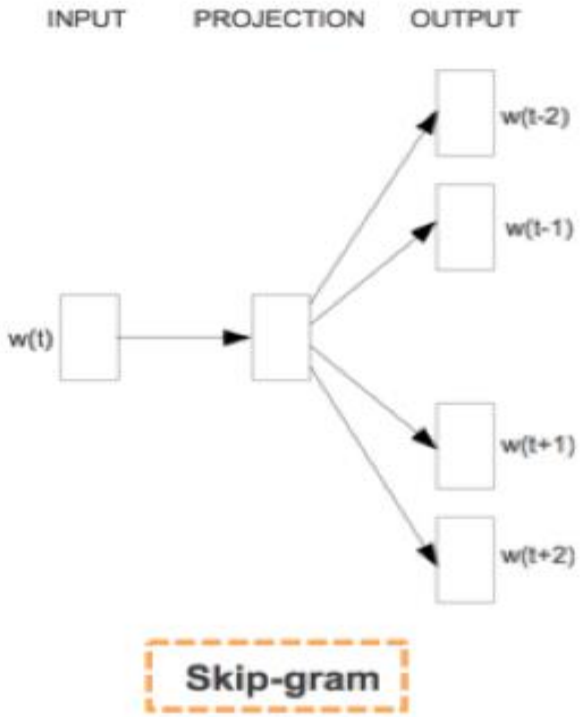
5. Word2Vec

- CBOW vs. Skip-gram



CBOW

문맥을 통해 현재 단어를 예측



Skip-gram

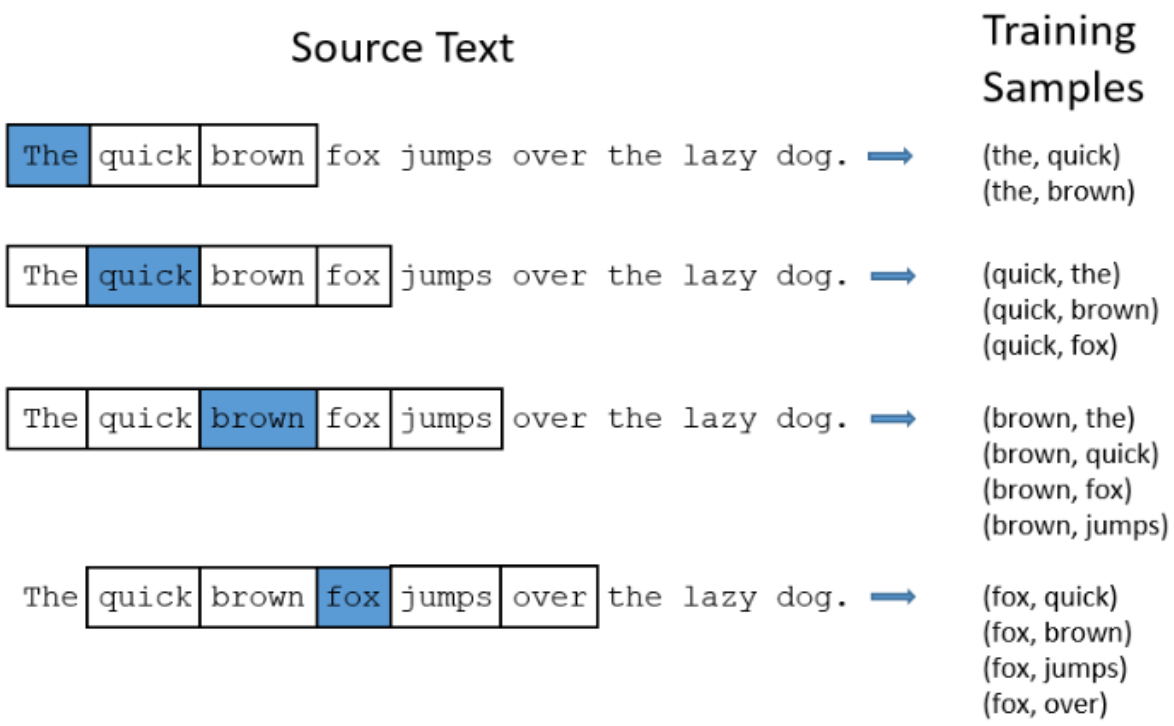
현재 단어를 통해 주위 문맥을 예측

Skip-gram은 크기가 큰 데이터 셋에 적합하므로 최근에 주로 사용

Growth Hackers

5. Word2Vec

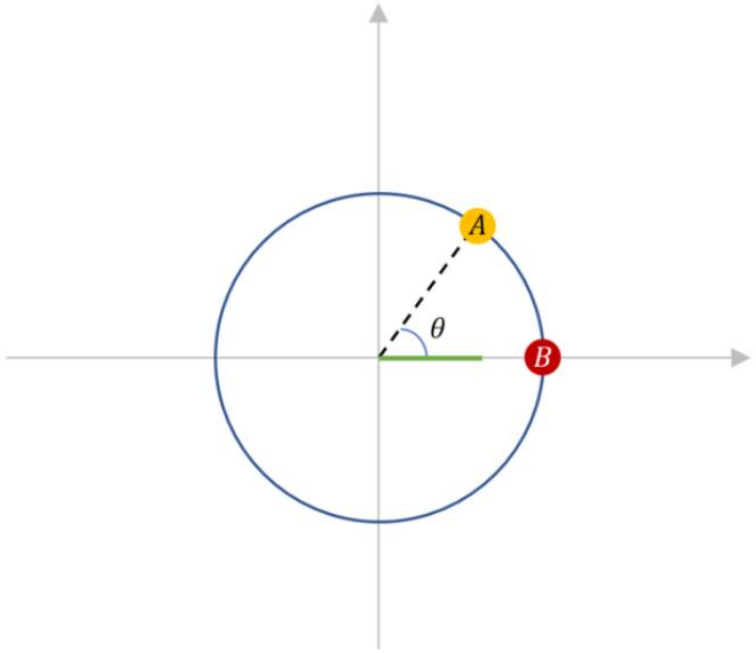
- Skip-gram
  - 윈도우(한번에 학습할 단어 개수) 크기가 2인 경우(예시)



Window 내에 등장하지 않는 단어에 해당하는 벡터는 중심단어 벡터와 벡터공간상에서 멀어지게, 등장하는 주변 단어 벡터는 중심단어벡터와 가까워지게 한다.

5. Word2Vec

- 코사인 유사도



코사인의 정의에 의해 cos(θ)는 그림의 녹색 선의 길이

cos(θ)는 단위원 내 벡터들끼리의 내적과 같음

내적이 커진다는 것은 두 벡터가 이루는 각 θ가 작아진다는 의미
즉 유사도가 높아진다는 의미!

5. Word2Vec

- Word2Vec 수식

$$p(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w=1}^{W} exp(u_w^T v_c)}$$

Word2Vec은 이 식을 최대화 하는 것을 목표로 함

식의 좌변은 중심단어(c)가 주어졌을 때 주변 단어(o)가 등장할 조건부 확률
-> 이 식을 최대화하는 것은 중심 단어로 주변 단어를 잘 맞춘다는 의미!

우변을 최대화한다는 것은 분자는 증가시키고, 분모를 줄이자는 것.

분자를 증가시킨다는 것은 벡터들 사이의 θ를 줄여서(**유사도가 높아진다**) 두 벡터의 내적값을 증가시킨다는 것이고, 분모는 중심 단어와 학습 말뭉치 내 모든 단어를 각각 내적한 것의 총합이므로 분모를 줄이려면 주변에 등장하지 않은 단어에 해당하는 벡터와 중심단어 벡터 사이의 θ 를 키워서(**유사도를 줄인다**) 내적값은 줄이자는 것.

5. Word2Vec

- Word2Vec 구현

```
# Word2Vec 모델 만들기
from gensim.models.word2vec import Word2Vec
model = Word2Vec(sentences,size=100, window=3, min_count=2, sg=1)
model.init_sims(replace=True)
```

Pos tagging된 컨텐츠를 100차원의 벡터로 바꿔라.
주변 단어(window)는 앞뒤로 세개까지 봐라.
Corpus내 출현 빈도가 2번 미만인 단어는 분석에서 제외해라
분석 방법론은 CBOW와 Skip-gram 중 후자를 선택해라

5. Word2Vec

- Word2Vec 구현

```
model.most_similar("hero")
```

```
[('heroine', 0.7999744415283203),
 ('organization', 0.7714229822158813),
 ('fictional', 0.7597637176513672),
 ('coach', 0.7547398209571838),
 ('hapless', 0.7482643723487854),
 ('estate', 0.7443069219589233),
 ('blonde', 0.7431157827377319),
 ('perspective', 0.7427940964698792),
 ('suit', 0.7416900396347046),
 ('companion', 0.7416478991508484)]
```

most_similar 함수는 두 벡터 사이의 코사인 유사도를 구해줌
해당 단어와 가장 비슷한(코사인 유사도가 큰) 단어를 출력

Quest

1. KoNLPy의 Kkma 태그를 이용하여 텍스트에서 가장 많이 쓰인 명사 10개 뽑기
   (각자 원하는 텍스트를 불러오시면 됩니다.)
2. LDA 코드를 활용하여 본인이 원하는 텍스트를 넣어 보기

   => 텍스트+결과 첨부+결과 해석

3. Word2Vec 코드를 활용하여 본인이 원하는 텍스트 넣어보기

   => 텍스트 + 결과 첨부

참고자료

- https://www.lucypark.kr/courses/2015-ba/text-mining.html#topic-modeling
- https://www.lucypark.kr/courses/2015-dm/text-mining.html (konlpy, nltk)
- https://ratsgo.github.io/from%20frequency%20to%20semantics/2017/06/01/LDA/ (LDA)
- https://ratsgo.github.io/from%20frequency%20to%20semantics/2017/07/09/lda/ (LDA 구현)
- https://ratsgo.github.io/from%20frequency%20to%20semantics/2017/03/30/word2vec/ (Word2Vec)