출처가 명시되지 않은 모든 자료(이미지 등)는 조성현 강사님의 강의 자료 바탕

문서 정보 추출

기계(컴퓨터)가 텍스트 문서로부터 특정 질문에 대한 정보를 추출할 수 있게 하는 것을 **Information Extraction**이라 한다.

인공지능 자연어처리 분야에서 큰 목표 중 하나로, 사람이 준 질문에 대해 기계가 문장의 내용을 이해해서 답변을 주도록 한다. 그러나 기계가 아직 사람처럼 문서의 이름을 이해하기는 어려우므로, 특정 패턴에 대한 정보만을 추출한다.

수업에서는 조직의 이름, 소재 위치 등에 대한 패턴을 추출하는 방법을 다룬다. 이를 위해서는 1) 이름 개체명 인식(*Name Entity Recognition*), 2) 관계 추출(*Relation Detection*) 이 필요하다.

일반적으로 다음의 과정을 거쳐 이루어진다.

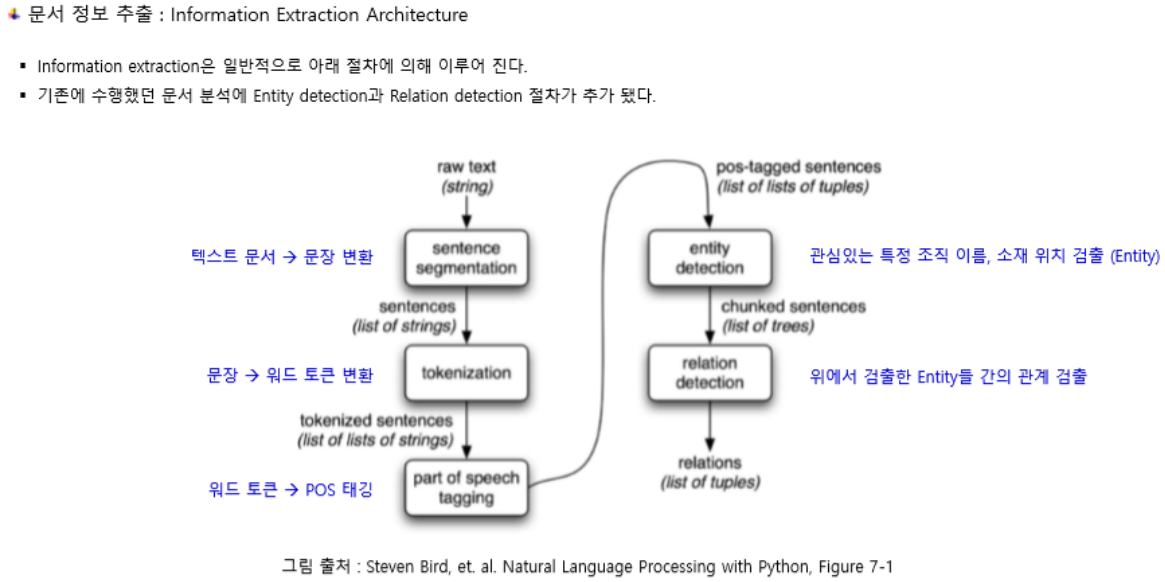


그림 출처 : Steven Bird, et. al. Natural Language Processing with Python, Figure 7-1

raw 텍스트 전처리 과정을 거친 뒤, 텍스트를 문장, 단어 토큰으로 변환한다. 이후 품사를 태깅한 후, 특정 조직, 이름, 소재, 위치 등을 검출한다.

1. 전처리 및 추출: Regular Expression

> 텍스트 전처리 및 필요한 정보 검출을 위한 필수 과정.

문장 내에서 정보를 추출하기 위해 필요한 문자열의 형태를 뽑아내야 한다. 정규표현식을 사용한다. 대표적으로 사용되는 정규표현식 operator는 다음과 같다.

- `.` : 아무 문자.
- `^` : 뒤에 나오는 패턴으로 시작.
- `$` : 앞에 나오는 패턴으로 끝.
- `[]` : 대괄호 안에 있는 것 중 하나.
- `*` : 앞에 나온 패턴이 없거나 있는 것.
- `+` : 앞의 패턴이 반드시 하나 이상 나와야 함.
- `{}` : 글자 수, 숫자 개수 지정.
- `\` : 이스케이프.

파이썬에서는 `re` 모듈을 활용한다.

```
import re
import nltk

treebank_words = nltk.corpus.treebank.words()
vowels = [v for w in treebank_words for v in re.findall(r'[aeiou]{2, }', w)] # 2개 이상 같이 쓰인 모음
fd_vowels = nltk.FreqDist(vowels)
print(fd_vowels.most_common(12))

# [('io', 549), ('ea', 476), ('ie', 331), ('ou', 329), ('ai', 261), ('ia', 253), ('ee', 217), ('oo', 174), ('ua', 109), ('au', 106), ('ue', 105), ('ui', 95)]
```

파이썬 Regex에서 `r`의 의미

기본적으로 문자열 앞에 `r`을 붙이면 이스케이프를 방지한다. `r'a\b'`는 실제로 `a\\b`와 같다.

이제 각 경우 별로 콘솔 창과 `print` 문을 사용해서 출력했을 때의 차이를 이해해 보자.

콘솔 창에 찍는 건 기본적으로 파이썬이 내부적으로 이해하는 바이다. `print`를 사용하면 제어문자까지 처리해서 보여주는 것이다.

```
word = r'back\\\slash$$'
>>> print(word) # back\\\slash$$
>>> print(r'back\\\slash$$') # back\\\slash$$
>>> word # 'back\\\\\\slash$$'
>>> r'back\\\slash$$' # 'back\\\\\\slash$$'
```

`print`를 통해 출력하면 `back\\\slash$$`가 나오지만, 콘솔 창에 그냥 찍으면 `back\\\\\\slash$$`와 같이 나온다. `r`을 붙였을 때 어떻게 이스케이프를 방지함으로써 `\`를 2개로 인식하는 것이다.

```
import re
>>> print(re.findall(r'\\', word)) # ['\\', '\\', '\\']
```

따라서 위의 문자열에서 `\\`를 찾으면, `back\\\\\\slash$$`에서 `\\`를 찾는 것과 동일한 결과가 난다.

```
>>> print(re.findall('\\\\', word))
```

위와 같은 의미이다.

```
word = 'back\\\slash$$t'
>>> print(word) # back\\slash$$t
>>> word # 'back\\\\slash$$t'
>>> print(re.findall(r'\$\$', word)) # ['$$']
>>> print(re.findall('\$\$', word)) # ['$$']
>>> print(re.findall('$$', word)) # ['']
```

위와 같이 문자열을 바꿔 주면, `r`이 붙지 않았기 때문에 맨 앞의 `\`가 이스케이프 문자가 된다. 따라서 `print`를 통해 출력하면 `\`가 2개밖에 없다. 콘솔 창에서 어떻게 출력되는지를 보면, 파이썬 내부에서는 `\\\\` 로서, 이스케이프 문자를 붙여 `\` 2개로 인식하고 있다.

`$`의 경우는 `r`을 붙여서 이스케이프를 해주든 해주지 않든 찾아오는 결과에 차이가 없다. 다만 맨 마지막에서처럼 `$$`를 찾으려고 하면, 찾지 못한다.

```
>>> re.findall(r"\\\\", r"\\") # ['\\\\']
```

위의 경우, `\\\\\\\\`에서 `\\\\`를 찾으라는 의미이다.

```
>>>> re.findall("\\\\", "\\") # ['\\']
```

위의 경우, `\\\\`에서 `\\`를 찾으라는 의미이다.

2. 구

Chunking

품사 태깅 후, 여러 개의 품사로 구(`Phrase`)를 만드는 것을 **Chunking**이라 한다. Chunking을 통해 만들어진 하나의 구를 **Chunk**라고 한다.

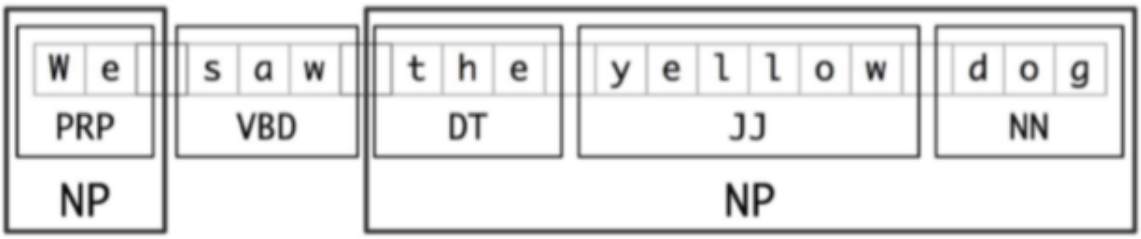


그림 출처 : Steven Bird, et. al. Natural Language Processing with Python, Figure 7-2

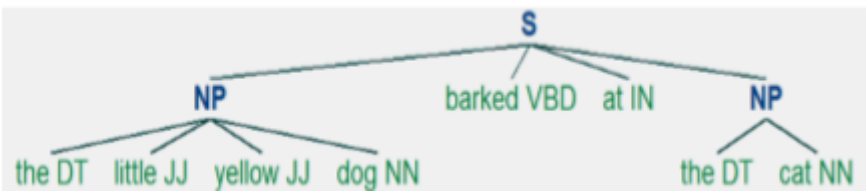
예컨대, 위의 그림에서 the yellow dog은 하나의 명사구(NP = DT+JJ+NN)를 이룬다.

`NLTK` 모듈에서 정규표현식을 이용해 문법을 정의한 뒤, chunk parser를 사용한다. `{regexp}`와 같은 식으로 표현한다.

```
import nltk

sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")] # 수동으로 POS-tagged 문장 생성.
grammer = "NP: {<DT>?<JJ>*<NN>}" # chunking 문법 정의
cp = nltk.RegexpParser(grammer) # chunk parser
result = cp.parse(sentence) # sentence 분석
print(result)
# (S
#   (NP the/DT little/JJ yellow/JJ dog/NN)
#   barked/VBD
#   at/IN
#   (NP the/DT cat/NN))
```

parser를 통해 파싱된 문장을 확인할 수 있다. chunking 문법에 맞는 구들이 chunk로 묶여 있다.

> *참고*
>
> `result.draw()`를 통해 그림을 그리려 했는데, `Colabaratory` 환경에서는 `TclError: no display name and no $DISPLAY environment variable`와 같은 에러가 난다.
>
> 해결해 보려 했는데 단시간에 되지는 않을 것 같아, 강사님의 결과 사진을 첨부하고 넘어 간다.
>
> 
>

그림에서도 확인할 수 있듯, 묶인 chunk는 tree 구조로 형성되어 있다. tree 구조 안에 있는 subtree도 확인할 수 있다. 묶여 있는 각각이 모두 subtree로 분류된다.

- `S` : 문장

- `NP`: 명사구

`print`를 통해 확인하면 tuple 형태로 묶여 있다. subtree의 `label`을 확인하여 문장 전체가 아니라 어떤 단위로 묶여 있는지 확인할 수 있다.

```
from pprint import pprint

sentence =  [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]

grammar = "NP: {<DT>?<JJ>*<NN>}" # <DT> 있어도 되고 없어도 되고, <JJ> 여러 개, <NN> 으로 끝나야 함.
parser = nltk.RegexpParser(grammar)
tree = cp.parse(sentence)
>>> pprint(tree)
# (S
#  (NP the/DT little/JJ yellow/JJ dog/NN)
#  barked/VBD
#  at/IN
#  (NP the/DT cat/NN))

# subtree 확인
for subtree in tree.subtrees():
    >>> print(subtree.label())
    # S
    # NP
    # NP
```

원하는 형태의 시퀀스만 발췌할 수도 있다.

- `<V.*>` : V로 시작하는 POS.
- 'CHUNK'라는 이름으로 묶어 준다.

```
# 원하는 형태의 시퀀스 발췌
cp = nltk.RegexpParser('CHUNK: {<V.*> <TO> <V.*>}')
brown = nltk.corpus.brown
i = 0
for sent in brown.tagged_sents(): # brown 코퍼스 POS 태깅
    tree = cp.parse(sent) # 파싱
    for subtree in tree.subtrees():
        if subtree.label() == "CHUNK":
            print(subtree)
            i += 1
    if i > 10:
        break
```

Chinking

Chunking의 반대로, 문장의 특정 부분을 chunk 밖으로 빼내는 것을 의미한다. chink rule은 `}regexp{` 이다.

문장에서 chunk를 제외한 나머지 부분으로, 문장 전체를 chunk로 정의하고, 특정 부분을 chinking하면 나머지 부분이 chunk가 된다. 이를 이용해 chinking을 이용해 chunking을 할 수도 있다.

```
grammar = r"""
    NP:
    {<.*>+}
    }<VBD|IN>+{
    """

sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"),
("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
cp = nltk.RegexpParser(grammar)
result = cp.parse(sentence)

>>> print(result)
# (S
#  (NP the/DT little/JJ yellow/JJ dog/NN)
#  barked/VBD
#  at/IN
#  (NP the/DT cat/NN))
```

- `NP:{<.*>+}`만 하면 모든 문장의 부분이 `NP`로 chunk가 된다.
- chinking rule을 통해 VBD나 IN을 chunk에서 빼내 준다.

IOB tags

chunk 안에 있는 각 품사의 위치에 따라 B(begin), I(inside), O(outside)로 구분한다.

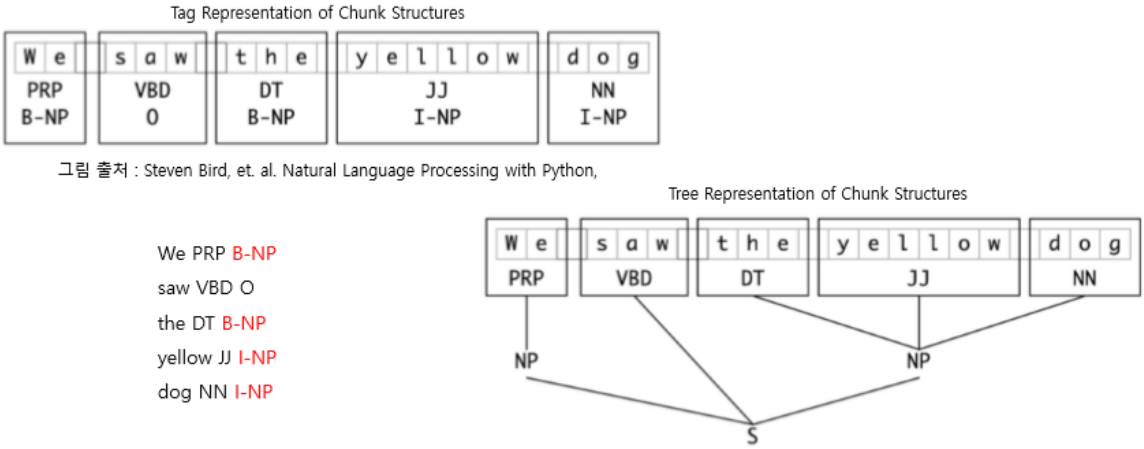


그림 출처 : Steven Bird, et. al. Natural Language Processing with Python,

위와 같은 문장이 있다고 하자. 'the yellow dog'이 하나의 명사구(NP)를 이룬다. 그 안에서 'the'는 NP의 시작인 `B-NP`가 되고, 'yellow'와 'dog'은 NP의 내부에 있으므로 `I-NP`가 된다.

3. 절

Chunking

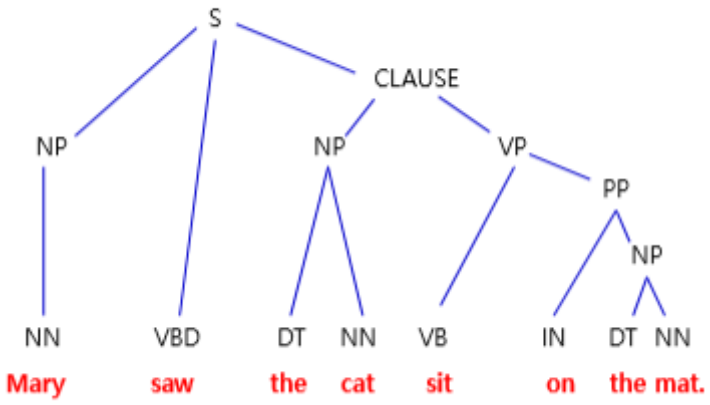
정규표현식 문법으로 절을 생성한다. 이후 파싱한다. 정의한 문법대로 문장을 분석할 수 있게 된다.

```
# CLAUSE 정의
grammar = r"""
    NP: {<DT|JJ|NN.*>+}
    PP: {<IN><NP>}
    VP: {<VB.*><NP|PP|CLUASE>+$}
    CLAUSE: {<NP><VP>}
    """

cp = nltk.RegexpParser(grammar)
sentence = sentence = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat",
"NN"), ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
tree = cp.parse(sentence)

>>> print(tree)
(S
  (NP Mary/NN)
  saw/VBD
  (CLAUSE
    (NP the/DT cat/NN)
    (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))
```

트리 구조를 그림으로 나타내 보면, 문장이 `Mary`와 `saw` 및 나머지 절로 이루어져 있음을 알 수 있다.



그러나 안은 문장, 안긴 문장 등 문법적 구조가 복잡한 경우, 트리를 깊게 구성해야 한다. 트리를 깊게 구성하지 않으면 절 안의 절이 제대로 분석되지 않아 clause 문법이 제대로 분석되지 않는다.

Parser 안에 `loop` 인자를 주어서 절 안의 절을 분석해야 한다. 아래의 예시를 잘못된 예와 잘된 예를 파악하자.

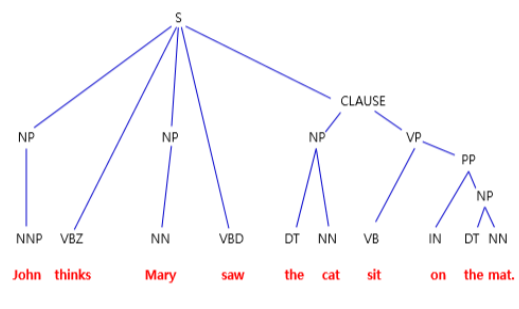
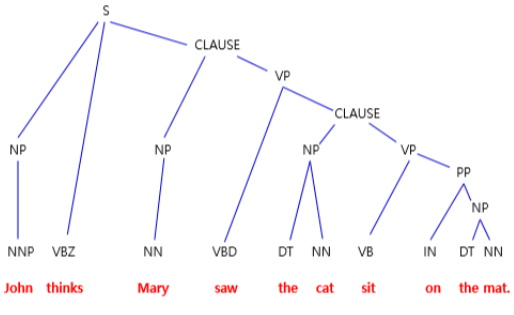
```
# 문장: "John thinks Mary saw the cat sit on the mat"

grammar = r"""
    NP: {<DT|JJ|NN.*>+}
    PP: {<IN><NP>}
    VP: {<VB.*><NP|PP|CLUASE>+$}
    CLAUSE: {<NP><VP>}
    """

# 잘못 분석된 예
cp = nltk.RegexpParser(grammar)

>>> print(cp.parse.sentence)
(S
  (NP John/NNP)
  thinks/VBZ
  (NP Mary/NN)
  saw/VBD
  (CLAUSE
    (NP the/DT cat/NN)
    (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))

# 잘된 분석의 예
cp = nltk.RegexpParser(grammar, loop=2)
>>> print(cp.parse(sentence))
(S
  (NP John/NNP)
  thinks/VBZ
  (NP Mary/NN)
  saw/VBD
  (CLAUSE
    (NP the/DT cat/NN)
    (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))
```

잘못된 예	잘된 예
	

4. Trees

트리 구조로 문서 정보를 추출해낼 수 있다.

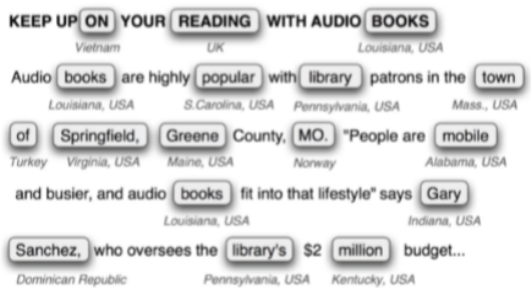
5. 개체명 인식

NLTK 패키지에서 `ne_chunk`를 사용하면 쉽게 사용할 수 있다. `binary=True` 옵션을 주면 개체명은 모두 `NE`로 나오고, `False`인 경우 `NE`도 그 안에서 분류되어서 나온다.

```
# NER
sent = nltk.corpus.treebank.tagged_sents()[22]

>>> print(nltk.ne_chunk(sent)) # binary 옵션 안 줬을 때
...
The/DT
(GPE U.S./NNP)
is/VBZ
...

>>> print(nltk.ne_chunk(sent, binary=True))
...
The/DT
(NE U.S./NNP)
is/VBZ
...
```

> *참고* : NER 구현의 어려움
>
> 문서 정보 추출 : Named Entity Recognition (NER) - 개체명 인식
>
> - Name entity type은 아래 그림과 같이 조직 이름 (ORGANIZATION), 사람 이름 (PERSON) 등으로 구분한다.
> - NER을 구현하는 것은 어려운 일이다. 아래 우측 그림은 일반 문장에 지명 사전 (Gazetteers)을 대입해서 location detection을 수행한 결과이다. 일반 단어들도 지명으로 분류하고 있다. 예를 들어 "ON"의 경우 베트남의 지명으로 분류하고 있다. Gazetteers (Geographical dictionary : 지명 사전) : 지역, 장소에 대한 지리적, 역사적, 문화적, 정치적, 경제적, 통계적 상황이나 인구 등에 대한 정보를 제공함.
> - 또한, 새로 생겨나는 이름들을 계속 업데이트하는 것도 관건이다.
>
> Commonly Used Types of Named Entity
>
> | NE Type | Examples |
> |---|---|
> | ORGANIZATION | *Georgia-Pacific Corp., WHO* |
> | PERSON | *Eddy Bonte, President Obama* |
> | LOCATION | *Murray River, Mount Everest* |
> | DATE | *June, 2008-06-29* |
> | TIME | *two fifty a m, 1:30 p.m.* |
> | MONEY | *175 million Canadian Dollars, GBP 10.40* |
> | PERCENT | *twenty pct, 18.75 %* |
> | FACILITY | *Washington Monument, Stonehenge* |
> | GPE | *South East Asia, Midlothian* |
>
> - Facility : human-made artifacts in the domains of architecture and civil engineering
> - GPE : entities such as city, state/province, and country
>
> 
>

개체명 인식 후 개체명 간 관계를 추출한다. 이를 **Relation Extraction**이라 한다.

```
import nltk
import re
from nltk.corpus import ieer

ieer = nltk.corpus.ieer
doc_ieer = ieer.parsed_docs('NYT_19980315')
print(doc_ieer[0].text[:20])

IN = re.compile(r'.*\bin\b(?!\b.+ing)')
for doc in doc_ieer:
    for rel in nltk.sem.extract_rels('ORG', 'LOC', doc,
                                     corpus='ieer', pattern=IN):
        >>> print(nltk.sem.rtuple(rel))


[ORG: 'WHYY'] 'in' [LOC: 'Philadelphia']
[ORG: 'McGlashan &AMP; Sarrail'] 'firm in' [LOC: 'San Mateo']
[ORG: 'Freedom Forum'] 'in' [LOC: 'Arlington']
[ORG: 'Brookings Institution'] ', the research group in' [LOC: 'Washington']
[ORG: 'Idealab'] ', a self-described business incubator based in' [LOC: 'Los
Angeles']
[ORG: 'Open Text'] ', based in' [LOC: 'Waterloo']
[ORG: 'WGBH'] 'in' [LOC: 'Boston']
[ORG: 'Bastille Opera'] 'in' [LOC: 'Paris']
[ORG: 'Omnicom'] 'in' [LOC: 'New York']
[ORG: 'DDB Needham'] 'in' [LOC: 'New York']
[ORG: 'Kaplan Thaler Group'] 'in' [LOC: 'New York']
[ORG: 'BBDO South'] 'in' [LOC: 'Atlanta']
[ORG: 'Georgia-Pacific'] 'in' [LOC: 'Atlanta']
```

- `re.compile(r'.*\bin\b(?!\b.+ing)')`
  - `\b` : blank.
  - `?!` : if not
  - `in` : 말 그대로 in.
- `extract_rels` : 관계 추출.