[2주차] KUBIG 22-1_NLP

✔️ **NLTK?**
: 자연어처리를 위한 파이썬의 대표적인 패키지

```
!pip install nltk
import nltk
nltk.download()
```

텍스트 전처리

: 풀고자 하는 문제와 사용하고자 하는 모델의 용도에 맞게 텍스트를 사전에 처리하는 작업

토큰화 (Tokenization)

- **토큰** : 의미를 갖는 문자열
  (= 형태소(뜻을 가진 최소 단위)나 그보다 상위 개념인 단어(자립해 쓸 수 있는 최소 단위)까지 포함하는 개념)
- **토큰화** : 문서나 문장을 분석하기 좋도록 의미있는 토큰 단위로 분리하는 작업

1. 단어 토큰화 (word tokenization)

: 토큰의 기준이 단어가 되는 토큰화 기법

- 영어 → 대부분 공백 단위로 단어 토큰화 가능
- 한국어 → 띄어쓰기가 더 복잡한 과정을 따르기 때문에 단순하게 공백 단위로만 토큰화하기에는 어려움

영어 단어 토큰화

1. NLTK 패키지의 `word_tokenize` 사용
   - ' 를 맥락과 함께 분리함

```
from nltk.tokenize import word_tokenize
print('단어 토큰화1 :',word_tokenize("Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."))
```

```
단어 토큰화1 : ['Do', "n't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr.', 'Jone', "'s", 'Orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'go
```

2. NLTK 패키지의 `WordPunctTokenizer` 사용
   - ' 를 별도로 하나의 토큰으로 분류함

```
from nltk.tokenize import WordPunctTokenizer
print('단어 토큰화2 :',WordPunctTokenizer().tokenize("Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."))
```

```
단어 토큰화2 : ['Don', "'", 't', 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr', '.', 'Jone', "'", 's', 'Orphanage', 'is', 'as', 'cheery', 'as', '
```

3. Keras의 `text_to_word_sequence` 사용
   - 모든 알파벳을 소문자로 바꿈, 구두점 제거, ' 은 보존함

```
from tensorflow.keras.preprocessing.text import text_to_word_sequence
print('단어 토큰화3 :',text_to_word_sequence("Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop."))
```

```
단어 토큰화3 : ["don't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', 'mr', "jone's", 'orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'goes', 'for', 'a
```

즉, 같은 입력 문장이라고 하더라도 어떤 토크나이저를 사용하느냐에 따라 단어 토큰화가 다르게 진행된다 ! 주어진 데이터에 대해서 모든 옵션을 적용해보고, 해당 데이터에 가장 적합한 토크나이저를 선택해 사용한다 !

✔️ **단어 토큰화시에 고려해야 할 사항:**
1. 구두점이나 특수 문자를 단순 제거하면 안된다.
2. 줄임말과 단어 내에 띄어쓰기가 있는 경우에 유의해야 한다.
예) we're이나 what're과 같이은 줄임말 / rock'n'roll과 같은 단어 → 하나의 토큰으로 인식

🔺 표준 토큰화 방식 = `Penn Treebank Tokenization`
1. 하이픈(-)으로 구성된 단어를 하나로 유지
2. doesn't와 같이 아포스트로피(')로 접어가 함께하는 단어는 분리

```
from nltk.tokenize import TreebankWordTokenizer
tokenizer=TreebankWordTokenizer()
text="Starting a home-based restaurant may be an ideal. it doesn't have a food chain or restaurant of their own."

print(tokenizer.tokenize(text))
```

```
['Starting', 'a', 'home-based', 'restaurant', 'may', 'be', 'an', 'ideal.',
'it', 'does', "n't", 'have', 'a', 'food', 'chain', 'or', 'restaurant', 'of', 'their', 'own', '.']
```

2. 문장 토큰화 (sentence tokenization)

: 토큰의 기준이 문장이 되는 토큰화 기법 → 코퍼스 내 문장 단위로 구분하는 작업, **문장 분류**(sentence segmentation)이라고 불림.

영어 문장 토큰화

1. NLTK 패키지의 `sent_tokenize` 사용

```
from nltk.tokenize import sent_tokenize

text = "His barber kept his word. But keeping such a huge secret to himself was driving him crazy. Finally, the barber went up a mountain and almost to the edge of
print('문장 토큰화1 :',sent_tokenize(text))
```

```
문장 토큰화1 : ['His barber kept his word.', 'But keeping such a huge secret to himself was driving him crazy.', 'Finally, the barber went up a mountain and almost to
```

   - 마침표가 여러번 등장하는 예시에도 robust한 기법임.

예) "I am actively looking for Ph.D. students. and you are a Ph.D student." → ['I am actively looking for Ph.D. students.', 'and you are a Ph.D student.']

한국어 문장 토큰화

1. `KSS` (Korean Sentence Splitter) 사용

```
!pip install kss
import kss
text = '쿠빅 자연어 처리 분반 화이팅입니다! 두 달동안 같이 열심히 해봐요!'
print('한국어 문장 토큰화 :',kss.split_sentences(text))
```

```
한국어 문장 토큰화 : ['쿠빅 자연어 처리 분반 화이팅입니다!', '두 달동안 같이 열심히 해봐요!']
```

마침표 처리 기법

위에서 보다싶이 문장 토큰화에서 예외 사항인 **마침표**를 처리하기 위해서는 입력에 따라 2개의 클래스로 분류해주는 이진 분류기(binary classifier)도 사용해볼 수 있다.

1. 마침표가 단어의 일부분인 경우 → 약어(abbreviation)으로 쓰이는 경우

> Abbreviations | Oxford English Dictionary
> We use cookies to enhance your experience on our website. By clicking 'continue' or by continuing to use our website, you are agreeing to our use of cookies. You can change your cookie settings at any time. Jump to Content This list contains the most common abbreviations used in the OED.
> https://public.oed.com/how-to-use-the-oed/abbreviations/

2. 마침표가 정말 문장의 구분자(bounary)인 경우

한국어 토큰화의 어려움

보시다싶이, 한국어 토크나이저는 영어에 비해서 더 적고, 토큰화 자체가 어려운 편입니다. 이유는,

1. **교착어의 특성**

영어에서처럼 띄어쓰기 단위가 독립적인 단어라면 띄어쓰기 단위로 토큰화가 가능하지만, 한국어는 어절이 독립적인 단어로만 구성되는 것이 아니라 조사 등의 무언가가 붙어있는 경우가 많다. → 모두 분리해줄 필요가 있음
예) 그는, 그에게, 그를, 그와 등을 모두 다른 단어로 취급?

→ **한국어의 경우 '형태소 분석'을 한다고 표현함 !**

- **형태소**(morpheme) : 뜻을 가진 가장 작은 말의 단위
  - 자립 형태소 - 접사, 어미, 조사와 관계없이 자립해 사용될 수 있는 형태소 (그 자체로 단어가 됨)
  - 의존 형태소 - 다른 형태소와 결합해 사용되는 형태소

예) 문장: '은지가 책을 읽었다.' → ['은지가', '책을', '읽었다']

- 자립 형태소 - 은지, 책
- 의존 형태소 - -가, -을, 읽-, -었, -다

2. **띄어쓰기의 모호성**

한국어서는 영어에서와 같이 띄어쓰기가 잘 지켜지지 않는 경향이 있다.

형태소 분석 / 품사 태깅 (POS tagging)

한국어 형태소 분석은 대부분 '`KoNLPy`' 패키지를 사용함.
품사 태깅을 하는 이유는 특정 토큰이 어떤 품사로 쓰였는지를 구분하기 위함임.

형태소 분석을 하는 이유?

1. 형태소 단위로 의미있는 단어를 가져가기 위함
2. 품사 태깅을 통해서 형용사나 명사를 추출하기 위함

예) 기존처럼 띄어쓰기 단위로 토큰화를 하게 되면, '기다연은', '기다연이', '기다연을'은 모두 다른 조사가 붙었다는 이유로 서로 다른 단어 벡터를 갖게 된다.

- 기다연은 - [1,0,0]
- 기다연이 - [0,1,0]
- 기다연을 - [0,0,1]

하지만 형태소 분석을 통해 '기다연'이라는 토큰을 추출한다면, 앞의 세 단어는 모두 동일한 벡터를 갖게 되어 모델이 더 효율적으로 학습할 수 있도록 해준다.

형태소 분석기의 종류

1. Kkma()

- `kkma.morphs` - 형태소 단위
- `kkma.nouns` - 명사 단위
- `kkma.pos` - 품사 태깅

```
from konlpy.tag import Kkma
kkma = Kkma()

sentence = '데이콘에서 다양한 컴피티션을 즐기면서 실력있는 데이터 분석가로 성장하세요'
print("형태소 단위 : ", kkma.morphs(sentence))
print("명사 단위 : ", kkma.nouns(sentence))
print("품사 태깅 : ", kkma.pos(sentence))
```

- 형태소 단위 : ['데이', '콘', '에서', '다양', '하', 'ㄴ', '컴피티션', '을', '즐기', '면서', '실력', '있', '는', '데이터', '분석가', '로', '성장', '하', '세요']
- 명사 단위 : ['데이', '데이콘', '콘', '다양', '컴피티션', '실력', '데이터', '분석가', '성장']
- 품사 태깅 : [('데이', 'NNG'), ('콘', 'NNG'), ('에서', 'JKM'), ('다양', 'NNG'), ('하', 'XSV'), ('ㄴ', 'ETD'), ('컴피티션', 'UN'), ('을', 'JKO'), ('즐기', 'VV'), ('면서', 'ECE'), ('실력', 'NNG'), ('있', 'VV'), ('는', 'ETD'), ('데이터', 'NNG'), ('분석가', 'NNG'), ('로', 'JKM'), ('성장', 'NNG'), ('하', 'XSV'), ('세요', 'EFN')]

2. Okt()

- `Okt.morphs` - 형태소 단위
- `Okt.nouns` - 명사 단위
- `Okt.pos` - 품사 태깅

```
from konlpy.tag import Okt
okt = Okt()

sentence = '데이콘에서 다양한 컴피티션을 즐기면서 실력있는 데이터 분석가로 성장하세요'
print("형태소 단위 : ", Okt.morphs(sentence))
print("명사 단위 : ", Okt.nouns(sentence))
print("품사 태깅 : ", Okt.pos(sentence))
```

3. Mecab()

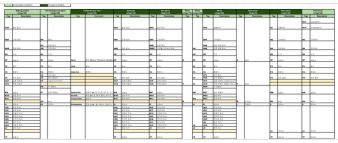
- `Mecab.morphs` - 형태소 단위
- `Mecab.nouns` - 명사 단위
- `Mecab.pos` - 품사 태깅

형태소 분석기 비교

한국어 POS 태깅 비교표

chart Not provided in KoNLPy,Provided in KoNLPy Sejong project (ntags=42),Sim Gwangsub project (ntags=26),Twitter Korean Text (ntags=19),Komoran (ntags=42),Mecab-ko (ntags=43),Kkma (ntags=10),Kkma (ntags=30),Kkma (ntags=56),Hannanum (ntags=9),Hannanum (ntags=22),Hannanum (ntags=26) Tag,Descripti...

https://docs.google.com/spreadsheets/d/1yUgThv3M0_zM_7uYD2xat7i5tOyEPUO4L46lYN3Rvfl/edit?usp=sharing



1. **Kkma** → 분석 시간이 매우 오래 걸림 (잘 사용하지 않음)
2. **Okt** → 품사 태깅 결과를 NNG나 XSV가 아닌, Noun, Verb와 같이 가독성 좋게 변환해줌
3. **Mecab** → 속도가 매우 빠름
4. **Komoran** → 오탈자를 어느정도 고려해줌
5. **khaiii** → 카카오에서 공개한 분석기로 성능이 좋음

이외에도 **Twitter**, **Hannanum** 등의 다양한 형태소 분석기가 존재합니다 ! 이것도 토크나이저와 동일하게 자신이 사용하는 데이터에 알맞게 사용해보는 것을 추천합니다 !

다음으로 소개할 어간 추출 + 표제어 추출은 모두 **정규화(normalization)** 기법으로, 코퍼스 내 비슷한 단어를 일반화시켜 총 단어의 개수를 줄일 수 있는 방법이다.

어간 추출 (stemmaization)

: 단어를 축약형으로 바꿔주는 작업

예) am → am, has → ha, watched → watch, doing → do

- NLTK 패키지의 `PorterStemmer` 사용
  - 포터 알고리즘 규칙: ALIZE → AL, ANCE → 제거, ICAL → IC
  - 위 규칙에 따르게 되면, formalize → formal, allowance → allow, electrical → electric
- NLTK 패키지의 `LancasterStemmer` 사용

```
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer

porter_stemmer = PorterStemmer()
lancaster_stemmer = LancasterStemmer()
```

```
어간 추출 전 : ['policy', 'doing', 'organization', 'have', 'going', 'love', 'lives', 'fly', 'dies', 'watched', 'has', 'starting']
포터 스테머의 어간 추출 후: ['polici', 'do', 'organ', 'have', 'go', 'love', 'live', 'fli', 'die', 'watch', 'ha', 'start']
랭커스터 스테머의 어간 추출 후: ['policy', 'doing', 'org', 'hav', 'going', 'lov', 'liv', 'fly', 'die', 'watch', 'has', 'start']
```

표제어 추출 (lemmatization)

: 품사 정보가 보존된 형태의 기본형으로 변환하는 작업

예) am → be, has → have, watched → watch, doing → do

- NLTK 패키지의 `WordNetLemmatizer` 사용

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print('표제어 추출 전 :',words)
print('표제어 추출 후 :',[lemmatizer.lemmatize(word) for word in words])
```

```
표제어 추출 전 : ['policy', 'doing', 'organization', 'have', 'going', 'love', 'lives', 'fly', 'dies', 'watched', 'has', 'starting']
표제어 추출 후 : ['policy', 'doing', 'organization', 'have', 'going', 'love', 'life', 'fly', 'dy', 'watched', 'ha', 'starting']
```

다음으로 소개할 대/소문자 통합과 불필요한 단어 제거는 모두 **정제(cleaning)** 기법으로, 코퍼스로부터 노이즈 데이터를 제거해주고 일관성 있게 단어를 표현하기 위함이다.

노이즈 데이터/불용어 제거 (stopwords)

✔️ **노이즈 데이터?**
1. 등장 빈도가 적은 단어 (희귀 단어)
2. 길이가 짧은 단어 (통상적으로 영어=길이 ≤ 2~3, 한국어=길이 ≤ 1)

✔️ **불용어?**
: 분석에 큰 의미가 없는 단어 토큰 (실제로는 많이 등장하지만 큰 도움이 되지 않는 단어들)

영어 불용어 제거

- NLTK 패키지에서 사전 정의된 `stopwords`와 매칭해 제거

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

stop_words_list = stopwords.words('english')  # 179개의 영어 불용어

example = "Family is not an important thing. It's everything."
stop_words = set(stopwords.words('english'))

word_tokens = word_tokenize(example)

result = []
for word in word_tokens:
    if word not in stop_words:
        result.append(word)

print('불용어 제거 전 :',word_tokens)
print('불용어 제거 후 :',result)
```

```
불용어 제거 전 : ['Family', 'is', 'not', 'an', 'important', 'thing', '.', 'It', "'s", 'everything', '.']
불용어 제거 후 : ['Family', 'important', 'thing', '.', 'It', "'s", 'everything', '.']
```

- 길이가 짧은 단어 (2~3 이하인 단어들)

```
# 길이가 1~2인 단어들 정규 표현식으로 삭제
import re
text = "I was wondering if anyone out there could enlighten me on this car."
shortword = re.compile(r'\W*\b\w{1,2}\b')  # 1~2사이인 word

print(shortword.sub('',text))  # 공백처리
```

```
was wondering anyone out there could enlighten this car.
```

한국어 불용어 제거

: 간단하게 토큰화를 진행한 후에 불필요한 단어들 정의해서 제거

- 조사, 접속사 제거
- 사용자가 직접 불용어 사전을 정의해서 제거
  → 구글에 '한국어 불용어 리스트'를 검색하시고 xlsx로 다운받아, 해당 파일 내에 있는 모든 단어를 '불용어 사전'으로 이용할 수 있습니다 !

Korean Stopwords
아 휴 아이구 아이쿠 아이고 어 나 우리 저희 따라 의해 을 를 에 의 가 으로 로 에게 뿐이다 의거하여 근거하여 입각하여 기준으로 예하면 예를 들면 예를 들자면 저 소인 소생 저희 지말고 하지마 하지마라 다른 물론 또한 그리고 비길수 없다 해서는 안된다 뿐만 아니라 만이 아니다 만은 아니다
https://www.ranks.nl/stopwords/korean

한국어 불용어 리스트 100개
블로그 텍스트를 분석하는 과제를 하던 도중, 한국어 불용어를 제거해야할 일이 생겼어요. komoran으로 형태소 분석후에 어미나 조사는 싹 빼버렸지만, 명사/동사/형용사 등에서 불용어를 제거할 수 있는 좋은 방법이 없어서 웹을 검색했으나 원하는 리스트를 찾지 못했습니다. 그래서 그냥 가지고 태깅 한국어 코퍼스에서 고빈도어 상위 100개를 뽑아 불용어 리스트를 만들어 보았습니다. utf-8 형태에 탭으로 구분된 파일입니다.
https://bab2min.tistory.com/544

정규 표현식 (regular expression)

: 규칙이 있는 데이터에 대해서 규칙을 선언해 정제할 수 있는 방법

예) '40,000원'이라는 텍스트에서 쉼표와 '원'을 제거하고 '40000'만 남겨두고 싶다 ! → 정규 표현식 사용

정규 표현식 조합

1. ✅ 숫자만 가능 : [0~9] → 띄어쓰기 불가능 `/^[0-9]+$/`
2. 이메일 형식만 가능 : `/^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.[a-z]{2})?)$/`
3. ✅ 한글만 가능 : [가나다라 ...] `/^[가-힣]+$/`
4. ✅ 영문만 가능 : `/^[a-zA-Z]+$/`
5. 영문,띄어쓰기만 가능 : `/^[a-zA-Z\s]+$/`
6. 전화번호 형태 → 전화번호 형태 000-0000-0000만 가능 : `]/^[0-9]{2,3}-[0-9]{3,4}-[0-9]{4}$/`
7. 도메인 형태 → http:// https:// 포함여부 상관없음 : `/^(((http(s?))\:\/\/)?)([0-9a-zA-Z\-]+\.)+[a-zA-Z]{2,6}(\:[0-9]+)?(\/\S*)?$/`
8. 도메인 형태 → http:// https:// 꼭 포함 : `/^((http(s?))\:\/\/)([0-9a-zA-Z\-]+\.)+[a-zA-Z]{2,6}(\:[0-9]+)?(\/\S*)?$/`
9. 도메인 형태 → http:// https:// 포함하면 안됨 : `/^[^((http(s?))\:\/\/)]([0-9a-zA-Z\-]+\.)+[a-zA-Z]{2,6}(\:[0-9]+)?(\/\S*)?$/`
10. ✅ 한글과 영문만 가능 : `/^[가-힣a-zA-Z]+$/;`
11. ✅ 숫자와 영문만 가능 : `/^[a-zA-Z0-9]+$/;`

정규 표현식 문법

~로 시작/끝/포함하는 문자열

Aa 정규표현식	패턴
^The	The로 시작하는 문자열
of despair$	of despair로 끝나는 문자열
^abc$	abc로 시작하고 abc로 끝나는 문자열 (abc 라는 문자열도 해당됨)
notice	notice가 들어 있는 문자열

~ 다음에 문자열

Aa 정규표현식	패턴
ab*	a 다음에 b가 0개 이상 (a, ab, abbb 등등)
ab+	a 다음에 b가 1개 이상 (ab, abbb 등등)
ab?	a 다음에 b가 있거나 없거나 (ab 또는 a)
ab{2}	a 다음에 b가 2개 있는 문자열 (abb)
ab{2,}	a 다음에 b가 2개 이상 (abb, abbbb 등등)
ab{3,5}	a 다음에 b가 3개에서 5개 사이 (abbb, abbbb, 또는 abbbbb)

(): 문자열 묶음 처리 / |: 또는

Aa 정규표현식	패턴
a(bc)*	a 다음에 bc가 0개 이상 (묶음 처리)
a(bc){1,5}	a 다음에 bc가 1개에서 5개 사이
hi|hello	hi나 hello가 들어 있는 문자열
(b|cd)ef	bef 또는 cdef
(a|b)*c	a와 b가 섞여서 여러번 나타나고 그뒤에 c가 붙어있는 패턴

기타 표현식

Aa 정규표현식	패턴
.{점}	임의의 한 문자
^.{3}$	3문자로만 되어 있는 문자열
[.]	괄호 안에 있는 내용 중 임의의 한 문자
[^.]	첫문자로 ^를 쓰면 괄호 내용의 부정. 즉 괄호 안에 포함되지 않는 한 문자
[ab]	a 또는 b (a|b 와 동일한 표현)
[a-d]	소문자 a에서 d까지 (a|b|c|d 또는 [abcd] 와 동일)
^[a-zA-Z]	영문자로 시작하는 문자열
[0-9]%	% 문자 앞에 하나의 숫자가 붙어 있는 패턴
%[^a-zA-Z]%	두 % 문자 사이에 영문자가 없는 패턴

특수문자 자체를 검색/사용

Aa 정규표현식	패턴
\^	^
\[	[
\(	(
|	|

Aa 정규표현식	패턴
\+	+
\{	{
\n	줄넘김 문자
\w	알파벳과 _(언더바)
\s	빈 공간
\d	숫자
\b	단어와 단어 사이의 경계
\t	tab 문자
\.	.
\$	$
\)	)
*	*
\?	?
\\	\
\r	리턴 문자
\W	알파벳과 _가 아닌 것
\S	빈 공간이 아닌 것
\D	숫자가 아닌 것
\B	단어 사이의 경계가 아닌 것
\xnn	16진수 nn에 해당하는 문자

✅ 예시:

```
text = """100 John    PROF
101 James   STUD
102 Mac   STUD"""
```

```
re.findall('\d+',text)
```

re.findall() : 정규표현식과 매치되는 모든 경우 문자열로 찾아 리스트로 리턴
₩d : 숫자에 해당되는 정규표현식
+ : 최소 1개 이상인 경우
➡ 최소 1개 이상의 숫자를 갖는 문자열 찾아 리스트로 리턴

```
['100', '101', '102]
```

정규 표현식 모듈 함수

모듈 함수	설명
re.compile()	정규표현식을 컴파일하는 함수입니다. 다시 말해, 파이썬에게 전해주는 역할을 합니다. 찾고자 하는 패턴이 빈번한 경우에는 미리 컴파일해놓고 사용하면 속도와 편의성면에서 유리합니다.
re.search()	문자열 전체에 대해서 정규표현식과 매치되는지를 검색합니다.
re.match()	문자열의 처음이 정규표현식과 매치되는지를 검색합니다.
re.split()	정규 표현식을 기준으로 문자열을 분리하여 리스트로 리턴합니다.
re.findall()	문자열에서 정규 표현식과 매치되는 모든 경우의 문자열을 찾아서 리스트로 리턴합니다. 만약, 매치되는 문자열이 없다면 빈 리스트가 리턴됩니다.
re.finditer()	문자열에서 정규 표현식과 매치되는 모든 경우의 문자열에 대한 이터레이터 객체를 리턴합니다.
re.sub()	문자열에서 정규 표현식과 일치하는 부분에 대해서 다른 문자열로 대체합니다.

유용한 정규 표현식

- 전자우편 주소: `/^[a-z0-9_+.-]+@([a-z0-9-]+\.)+[a-z0-9]{2,4}$/`
- URL: `/^(file|gopher|news|nntp|telnet|https?|ftps?|sftp):\/\/([a-z0-9-]+\.)+[a-z0-9]{2,4}.*$/`
- HTML 태그: `/\<(/?[^\>]+)\>/`
- 전화 번호: `/(\d{3}).*(\d{3}).*(\d{4})/`
  예) 123-123-2344 또는 123-1234-1234
- 날짜: `/^\d{1,2}\/\d{1,2}\/\d{2,4}$/`
  예) 3/28/2007 또는 3/28/07
- jpg, gif 또는 png 확장자를 가진 그림 파일명: `/([^\s]+(?=\.(jpg|gif|png))\.\2)/`
- 16 진수로 된 색깔 번호: `/#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?/`

- 적어도 소문자 하나, 대문자 하나, 숫자 하나가 포함되어 있는 문자열(8글자 이상 15글자 이하) → 많은 사이트에서 올바른 암호형식인지 확인하기 위한 용도: `/(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15}/`
- 주민번호 (-까지 포함): `/^(?:[0-9]{2}(?:0[1-9]|1[0-2])(?:0[1-9]|[1,2][0-9]|3[0,1]))-[1-4][0-9]{6}$/`

정수 인코딩 (integer encoding)

: 단어에 정수 인덱스를 부여해 컴퓨터가 처리하기에 용이하도록 하는 작업

```
{'barber': 1, 'secret': 2, 'huge': 3, 'kept': 4, 'person': 5}
```

1. **dictionary** 사용
   : 단어를 빈도수 순으로 정렬한 단어집합 생성 (빈도수가 높은 순으로 낮은 숫자부터 정수 인덱스 부여)
2. **Counter** 사용
   : 중복을 제거한 단어의 등장 빈도수를 출력해주는 기능 (빈도수가 높은 순으로 낮은 숫자부터 정수 인덱스 부여)

```
from collections import Counter
all_words_list = ['jinsu', 'dayeon', 'yeonjung', 'eunji', 'yoonjung', 'yebin', 'yoonjung', 'dayeon', 'eunji', 'jinsu', 'eunji']
vocab = Counter(all_words_list)
print(vocab)
```

```
Counter({'jinsu': 2, 'dayeon': 2, 'yeonjung': 1, 'eunji': 3, 'yoonjung': 2, 'yebin':1})
```

3. NLTK 패키지의 `FreqDist` 사용
   : 빈도수 계산 도구 (빈도수가 높은 순으로 낮은 숫자부터 정수 인덱스 부여)

```
from nltk import FreqDist
```

4. Keras의 `Tokenizer` 사용
   : 빈도수 기준으로 단어집합 생성

```
from tensorflow.keras.preprocessing.text import Tokenizer
```

패딩 (padding)

: 문자열 간의 길이가 다른 경우, 임의로 동일한 길이로 맞춰주는 기법

1. `numpy` 사용
   : 가장 긴 길이의 문장 길이(max_length) 출력 → max_length보다 작으면 빈자리 0으로 패딩
2. Keras의 `pad_sequences` 사용
   - padding = 'pre' → 앞의 빈 공간 0으로 채우기
   - padding = 'post' → 뒤의 빈 공간 0으로 채우기

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

원핫 인코딩 (one-hot encoding)

: 단어 집합의 크기를 벡터 차원으로 두고, 표현하고 싶은 단어의 인덱스는 1, 나머지 인덱스에는 0을 부여하는 단어의 벡터 표현 방식

- Keras의 `to_categorical` 사용
  → 모든 데이터를 정수 인덱스로 먼저 표현 (정수 인코딩)해준 다음, 원핫 벡터로 표현

```
from tensorflow.keras.utils import to_categorical
```

원핫 인코딩의 한계점

- 비효율적인 메모리 사용량
  : 원핫 인코딩으로 표현하게 되면, '기다연' : 6 → [0,0,0,0,0,0,1]으로 표현하는 방식이기 때문에 표현 단어의 개수가 늘어나고, 자연스럽게 벡터를 저장하기 위해 필요한 공간이 늘어나게 된다.
- 단어의 유사도 표현불가
  : 원핫 벡터만으로는 두 개 이상 단어간의 유사도를 표현하기 어려움 [0,0,0,0,0,0,1] [0,0,0,0,1,0,0]

→ 단점을 해결하기 위해 단어의 잠재의미를 벡터화하기 위한 다른 방법들이 존재합니다. (이는 다음주에서 다룰 예정 !)

- count 기반 벡터화 → LSA (잠재의미 분석)
- 예측 기반 벡터화 → word2vec, FastText 등
- count + 예측 기반 벡터화 → GloVe

Appendix (한국어의 유용한 전처리 패키지)

저도 항상 한국어 텍스트 데이터를 이용해서 전처리할 때면 아래 패키지들을 필수적으로 적용합니다 !

1. 한국어 **띄어쓰기** 교정 → `PyKoSpacing`

```
!pip install git+https://github.com/haven-jeon/PyKoSpacing.git

from pykospacing import Spacing
spacing = Spacing()
kospacing_sent = spacing(sentence)  # sentence=입력 문장
```

2. 한국어 **맞춤법** 교정 → `Py-Hanspell`
   : 맞춤법 + 띄어쓰기 자동으로 함께 교정해줌

```
!pip install git+https://github.com/ssut/py-hanspell.git

from hanspell import spell_checker
spelled_sent = spell_checker.check(sentence)  # sentence=입력 문장
hanspell_sent = spelled_sent.checked
```

3. **만약 사용하는 데이터에 비속어나 신조어가 많다면 !**
   → 위에서 소개해드린 Kkma, Hannanum과 같은 전통적인 형태소 분석기 대신 '`soynlp`'라는 토크나이저를 추천해드립니다 !

- 비지도 학습으로 단어 토큰화 진행 (데이터에서 빈도수가 높은 단어들 위주)
- 내부적으로 단어 점수 표를 생성해서 동작함 → 응집 확률(cohesion probability) + 브랜칭 엔트로피(branching entropy) 사용

```
!pip install soynlp
```

응집 확률 (cohesion probability)

> 내부 문자열이 얼마나 응집해 자주 등장하는지를 판단하는 척도

- 왼쪽에서 오른쪽으로 순서대로 각 문자열이 주어졌을 때, 다음 문자열이 나올 확률의 누적합
- 응집 확률 높을수록 문자열 sequence가 하나의 단어로 나타날 확률이 높다는 의미

$$\text{cohesion}(n) = \left(\prod_{i=1}^{n-1} P(c_{1:i+1}|c_{1:i})\right)^{\frac{1}{n-1}}$$

- $cohesion(2) = P(\text{반포한}|\text{반})$
- $cohesion(3) = \sqrt{(P(\text{반포한}|\text{반}) \cdot P(\overline{\text{반포한}|\text{반포}}))}$
- $cohesion(4) = \sqrt{(P(\text{반포한}|\text{반}) \cdot P(\overline{\text{반포한}|\text{반포}}) \cdot P(\overline{\text{반포한강}|\text{반포한}}))}$
- $cohesion(5) = \sqrt{(P(\text{반포한}|\text{반}) \cdot P(\overline{\text{반포한}|\text{반포}}) \cdot P(\overline{\text{반포한강}|\text{반포한}}) \cdot P(\overline{\text{반포한강공}|\text{반포한강}}))}$
- $cohesion(6) = \sqrt{(P(\text{반포한}|\text{반}) \cdot P(\overline{\text{반포한}|\text{반포}}) \cdot P(\overline{\text{반포한강}|\text{반포한}}) \cdot P(\overline{\text{반포한강공}|\text{반포한강}}) \cdot P(\overline{\text{반포한강공원}|\text{반포한강공}}))}$
- $cohesion(7) = \sqrt{(P(\text{반포한}|\text{반}) \cdot P(\overline{\text{반포한}|\text{반포}}) \ldots \text{중략} \ldots \cdot P(\overline{\text{반포한강공원}|\text{반포한강공}}) \cdot P(\overline{\text{반포한강공원에}|\text{반포한강공원}}))}$

```
word_score_table["반포한"].cohesion_forward  # 0.0883
word_score_table["반포한강공원"].cohesion_forward  # 0.379
```

브랜칭 엔트로피 (branching entropy)

> 주어진 문자열에서 얼마나 다음 문자가 등장할 수 있는지를 판단하는 척도

- 주어진 문자열에서 다음 문자 예측을 위해 헷갈리는 정도로 비유됨 → 엔트로피 높을수록 헷갈리는 정도 높음 → 더 적은 정보가 주어졌다는 의미
- 하나의 완성된 단어에 가까워질수록 문맥으로 인해 점점 정확하게 예측이 가능해지며 엔트로피값이 줄어듦

```
word_score_table["디스"].right_branching_entropy  # 1.637
word_score_table["디스플"].right_branching_entropy  # -0.0 (레가 오는 것이 너무나도 명백하기 때문)
```