파이썬 예제

이전 데이터 수집 예제를 이어서 진행하겠습니다. 수집한 타임지 기사 본문에서 년도만 추출해보겠습니다.

```
import re
```

4자리의 숫자로 된 정규표현식 패턴을 compile 함수를 사용하여 생성합니다.

```
p = re.compile("[0-9]{4}")
```

정규표현식 객체를 가지고 있는 p의 findall 함수를 사용하여 타임지 뉴스 기사의 단락에서 4자리의 숫자로 된 값들만 추출합니다. findall 함수는 값이 없을 경우 빈 리스트를 반환하기 때문에 년도가 없는 단락은 출력을 제외하기 위해서 if문을 사용하였습니다.

```
for x in result.findAll("p"):
    rt = p.findall(x.text)
    if len(rt) > 0:
        print(rt)
```

```
['2023']
['2017', '2019', '2024']
['1994', '2003', '2008']
['1998']
['2014']
['2015', '2016']
['2021']
['2022', '2021']
['2020']
['2015', '2017']
['1990']
```

간단히 데이터 수집 및 정규 표현식을 사용하여 목적에 맞는 데이터를 추출해보았습니다.

02 텍스트 분석 역사 및 기법

자연어 처리 연구의 역사

NLP 연구의 역사는 크게 3가지로 구분할 수 있습니다. 규칙 기반 자연어 처리, 통계 기반 자연어 처리, 딥러닝 기반 자연어 처리 순서대로 발전되어 왔습니다.

규칙 기반이란 언어의 문법적인 규칙을 사전에 정의하고 그 규칙에 기반하여 자연어를 처리하는 방식을 의미합니다. 1954년 IBM과 조지타운 대학의 공동으로 러시아어-영어 번역기가 개발되었습니다. 이 번역기가 규칙 기반 자연어 처리의 시초라고 할 수 있습니다. 이 번역기의 데모에서 총 60개의 러시아어 문장을 영어로 번역해냈습니다. 1960년대에는 SHRDLU ELIZA와 같은 대화형 자연어 이해 시스템도 등장하였습니다.

기계번역을 규칙 기반으로 처리할 때는 핵심이 되는 단어들을 사전을 통해 번역한 다음, 원본 문장에서 발견되는 문법적인 규칙을 찾아낸 후 대응하는 규칙을 불러와 이를 이용해서 단어와 단어 사이를 이어주는 식으로 진행되었습니다.

규칙 기반 자연어 처리방법은 간단한 문장은 원활하게는 처리되었지만, 복잡하고 어려운 문장들은 사전에 구축된 규칙만으로는 해결되지 못하였습니다. 규칙을 추가적으로 추가하기 위해서 사람이 개입해서 만들어야 하기 때문에 시간과 자원이 많이 소모되었고, 그에 비해 언어가 가지고 있는 예외는 완벽하게 규칙을 만드는 게 불가능 했습니다.

또, 언어가 가지는 어순이나 특성 등으로 인하여 문장의 종류가 제한되거나 정확도가 매우 떨어지는 문제가 있습니다. 따라서 규칙 기반 처리 방식은 다른 처리 방법이 개발된 이후에 자연스럽게 사람들의 관심에서 멀어져 갔습니다.

규칙 기반 자연어 처리의 한계를 극복하기 위해서 제시된 방법이 바로 통계 기반 자연어 처리 방법입니다. 통계 기반의 핵심 개념은 조건부 확률에 있습니다. 단어를 예측할 때 앞 뒤에 등장하는 단어의 사이에 어떤 단어가 나올 확률이 가장 높은지를 계산할 수 있습니다. 이중 확률이 가장 높은 단어를 선택하면 자연스러운 문장이 될 가능성이 높아집니다.

언어 모델은 여러 개의 단어들이 동시에 출현할 확률이나 특정한 단어들이 주어질 때 그 다음 단어가 어떤 단어가 나올 지 예측하는데 사용되는 모형입니다. 규칙 기반 이후에 언어 모델은 통계 기반 언어 모델과 신경망 기반 언어 모델으로 발전하였습니다.

통계 기반은 형태소 사이의 유의미한 상관관계를 분석하는 것으로 시작했습니다. 컴퓨터의 성능이 발전하면서 대량의 문장을 빠르게 분석하는 것이 가능해지기 시작하면서 성과가 차차 나타나기 시작했습니다. 하지만 여전히 사람이 관여해야 하는 번거로움은 존재하였습니다. 이러한 문제는 딥러닝 기반 자연어 처리방법이 개발된 이후에 자연스럽게 딥러닝 기반 자연어 처리 방법으로 사람들의 관심이 이동했습니다.

현재 자연어 처리 기법은 신경망(딥러닝) 기법이 거의 모든 부분을 차지하고 있다고 해도 과언이 아닐 것입니다. 인공 신경망이라고 알려진 이 방식은 입력 데이터를 처리하고 가중치를 계산하며 다음 단계로 연결시키는 방법으로 고안된 알고리즘은 사람의 뇌의 신경계와 닮았다고 하여 붙여진 이름입니다.

신경망 기법은 Chat-GPT 같은 챗봇이나 생성형 AI에 큰 영향을 미쳤으며, 사람에 가까운 성능을 내고 있습니다. 조금 더 높은 성능을 높이기 위해서는 많은 말뭉치 및 계산을 진행할 수 있는 장비들이 필요하여 쉽게 만들기는 어렵다는 단점이 있습니다.

자연어 처리의 의의

자연어 처리(Natural Language Processing, NLP)는 사람의 언어를 컴퓨터가 이해 하거나 생성할 수 있도록 하는 분야입니다. 이를 바탕으로 NLP를 크게 언어 이해(NLU)와 언어 생성(NLG)로 2가지로 구분할 수 있습니다.

스마트폰에 음성으로 “영희에게 전화걸어줘”라고 말을 하면 전화 프로그램과 통화 대상인 영희의 연락처를 파악한 후 이것을 기반으로 영희에게 전화를 걸었다면 자연어를 잘 처리했다고 할 수 있다. 이런 과정을 NLU(Natural Language Understanding)라고 합니다.

인공지능이 글을 작성하거나, 챗봇이 사용자와 자연스러운 대화를 위해서 응답을 생성하는 것을 NLG(Natural Language Generation)이라고 합니다.

NLU와 NLG는 입출력이 서로 반대이기 때문에 처리 방법이 다릅니다. 하지만 실제 우리가 사용하는 인공지능 제품은 NLU와 NLG가 동시에 적용되고 있습니다.

Bag of Words

단어들의 순서는 고려하지 않고, 출현하는 단어의 빈도수(frequency)를 가지고 수치화 하는 방법입니다. 여기서 가방(bag)은 단어들이 있는 컨테이너를 의미하며, 이때 단어들의 순서나 구조를 고려하지 않고 단어의 출현 빈도에만 집중합니다.

scikit-learn은 파이썬에서 머신러닝을 진행할 수 있는 매우 유명한 라이브러리입니다. 해당 라이브러리에는 Bag of Words를 만들 수 있는 함수도 존재합니다. 해당 라이브러리를 설치하여 예제를 진행하겠습니다.

출력된 결과값의 가독성을 위해서 DataFrame형태로 출력하겠습니다. 이때 필요한 라이브러리가 pandas입니다. pandas도 같이 설치하겠습니다.

```
pip install scikit-learn
pip install pandas
```

scikit-learn 안에는 CountVectorizer, TfidVectorizer를 이용하여 단어의 빈도수를 계산합니다.

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

sentence_1= "You cannot be happy every day. But there are happy things every day"
CountVec = CountVectorizer()
Count_data = CountVec.fit_transform([sentence_1])

cv_dataframe=pd.DataFrame(Count_data.toarray(),columns=CountVec.get_feature_names_
out())
display(cv_dataframe)
```

	are	be	but	cannot	day	every	happy	there	things	you
0	1	1	1	1	2	2	2	1	1	1

위의 예제에서 불용어를 제거하고 진행하고 싶다면 아래처럼 CountVectorizer안에 stop_words를 사용하면 자체적으로 제공하는 불용어를 사용하여 제거합니다.

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

sentence_1= "You cannot be happy every day. But there are happy things every day"
CountVec = CountVectorizer(stop_words='english')
Count_data = CountVec.fit_transform([sentence_1])

cv_dataframe=pd.DataFrame(Count_data.toarray(),columns=CountVec.get_feature_names_
out())
display(cv_dataframe)
```

	day	happy	things
0	2	2	1

DTM(Docuement-Term Matrix)

사람이 사용하는 자연어는 기계가 이해할 수 없기 때문에 기계가 이해할 수 있는 숫자 나열인 벡터로 변환해야 합니다. 이런 과정의 산출물인 벡터 혹은 이런 과정 전체를 임베딩(Embedding)이라고 합니다.

임베딩이라는 개념을 아주 오래전부터 존재했지만, 자주 사용하기 시작한 시점은 요슈아 벤지오(Yoshua Bengio) 연구 팀의 논문 "A Neural Probabilistic Language Model"을 발표하고 난 이후 부터입니다.

아주 단순한 임베딩 기법부터 딥러닝을 활용한 word2vec까지 이번장에서 공부해보겠습니다.

아래 4개의 문장이 있습니다. 설명의 편의상 문장을 문서라고 지칭하겠습니다. 이 문서를 사용해서 등장한 모든 단어의 출현 빈도수를 행렬로 표현하겠습니다.

```
doc1 = "사과를 좋아한다. 바나나도 좋아한다"

doc2 = "사과 제품을 좋아한다. 여름에는 수박을 좋아한다"

doc3 = "배가 너무 고프다. 바나나를 먹고 싶다"

doc4 = "배가 고파 사과를 먹었다. 사과는 맛있다"
```

4개의 문서에서 명사만 추출해서 빈도수를 계산하면 아래와 같습니다.

	바나나	배	사과	수박	여름	제품
doc1	1	0	1	0	0	0
doc2	0	0	1	1	1	1
doc3	1	1	0	0	0	0
doc4	0	1	2	0	0	0

위와 같이 빈도표를 문서 단어 행렬(Docuement-Term Matrix)라고 부릅니다. 이렇게 DTM을 활용하면 문서별로 단어의 빈도수를 확인할 수 있습니다. 행렬의 각 행은 하나의 문서이며, 이 문서의 벡터값은 이 문서를 표현하는 숫자입니다. 이런 형태는 가장 간단한 형태의 임베딩입니다.

실제 현업에서는 위와 같은 임베딩 방법은 지금은 사용하지 않습니다. 복잡하고 정교한 임베딩 방식을 사용합니다.

TF-IDF

TF-IDF(Term Frequency - Inverse Document Frequency)는 정보 검색과 텍스트 마이닝에서 이용하는 가중치로, 여러 문서로 이루어진 문서군이 있을 때 어떤 단어가 특정 문서 내에서 얼마나 중요한 것인지를 나타내는 통계적 수치입니다. 문서의 핵심어를 추출하거나, 검색 엔진에서 검색 결과의 순위를 결정하거나, 문서들 사이의 비슷한 정도를 구하는 등의 용도로 사용할 수 있습니다.

TF(단어 빈도, term frequency)는 특정한 단어가 문서 내에 얼마나 자주 등장하는지를 나타내는 값으로, 이 값이 높을수록 문서에서 중요하다고 생각할 수 있습니다. 하지만 단어 자체가 문서군 내에서 자주 사용되는 경우, 이것은 그 단어가 흔하게 등장한다는 것을 의미합니다. 이것을 DF(문서 빈도, document frequency)라고 하며, 이 값의 역수를 IDF(역문서 빈도, inverse document frequency)라고 한다. TF-IDF는 TF와 IDF를 곱한 값입니다.

IDF 값은 문서군의 성격에 따라 결정된다. 예를 들어 '원자'라는 낱말은 일반적인 문서들 사이에서는 잘 나오지 않기 때문에 IDF 값이 높아지고 문서의 핵심어가 될 수 있지만, 원자에 대한 문서를 모아 놓은 문서군의 경우 이 낱말은 상투어가 되어 각 문서들을 세분화하여 구분할 수 있는 다른 낱말들이 높은 가중치를 얻게 됩니다.

TF-IDF 역시 단어 등장 순서를 고려하지 않는다는 점에서 Bag of Words 방식이라는 것을 알 수 있습니다.

TF-IDF의 수학적인 계산에 대해서 알아보겠습니다.

$$\mathrm{TF-IDF}(w) = \mathrm{TF}(w) \times \log\left(\frac{N}{\mathrm{DF}(w)}\right)$$

TF(Term Frequency)는 어떤 단어가 특정 문서에 얼마나 많이 사용되었는지를 나타내는 빈도를 의미합니다. 많이 사용되는 단어가 중요하다는 가정을 전체로 만든 수치입니다. 예를 들어 apple라는 단어가 doc1에서 5번, doc2에서 10번 쓰였다면 (doc1,apple)의 TF는 5, (doc2, apple)의 TF는 10이 됩니다.

DF(Document Frequency)란 특정 단어가 나타난 문서의 수를 의미합니다. apple이라는 단어가 말뭉치 전체에서 doc1, doc2, doc5에만 나타냈다면 DF는 3이 됩니다. DF가 클수록 많은 문서에서 사용되고 있는 일반적인 단어라고 생각할 수 있습니다. TF는 같은 단어라도 문서마다 다른 값을 갖고, DF는 문서가 달라지더라도 단어가 같다면 같은 값을 가지고 있습니다.

IDF(Inverse Document Frequency)는 전체 문서 수(N)를 해당 단어의 DF로 나눈 뒤 로그값을 취한 결과입니다. 그 값이 클수록 특이한 단어라는 의미입니다.

TF-IDF가 지향하는 원리는 어떤 단어의 주체 예측 능력이 강할수록 값이 커지고, 그 반대의 경우는 작게 나옵니다. 어떤 단어의 TF의 높게 나온다면 수식에 따라서 TF-IDF의 값도 크게 나옵니다.

높은 TF 값은 해당 단어가 해당 문서에서 중요하게 다뤄 졌음을 의미하며, 높은 IDF 값은 해당 단어가 특정 문서에서만 나타나고 다른 문서에서는 드물게 나타난다는 것을 의미합니다. 따라서 TF-IDF 값이 크게 나온다는 것은 특정 단어가 해당 문서에서 두 가지 측면에서 중요하다는 것을 나타냅니다.

해당 문서에서 자주 등장하고, 다른 문서에서는 드물게 등장하는 의미입니다. 이는 해당 단어가 해당 문서의 특징을 잘 나타내거나 문서의 주제에 중요한 영향을 미칠 수 있다는 것을 의미할 수 있습니다.

TF-IDF를 scikit-learn의 함수를 사용하여 아래와 같이 생성할 수 있습니다.

```
doc1 = “She love apple, You love apple.”
doc2 = “She hates snakes, and he likes bananas, but he also hates snakes.”
doc3 = “She like apple, snake”
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_trans = TfidfVectorizer()
tfidf_trans.fit([doc1, doc2, doc3])
pd.DataFrame(tfidf_trans.transform([doc1, doc2, doc3]).toarray(),
             columns=tfidf_trans.get_feature_names_out())
```

	also	and	apple	bananas	but	hates	he	like	likes	love	she	snake	snakes	you
0	0.000000	0.000000	0.549491	0.000000	0.000000	0.00000	0.00000	0.000000	0.000000	0.722515	0.213365	0.000000	0.00000	0.361258
1	0.240085	0.240085	0.000000	0.240085	0.240085	0.48017	0.48017	0.000000	0.240085	0.000000	0.141798	0.000000	0.48017	0.000000
2	0.000000	0.000000	0.444514	0.000000	0.000000	0.00000	0.00000	0.584483	0.000000	0.000000	0.345205	0.584483	0.00000	0.000000

Word Embedding

워드 임베딩(word embedding)은 단어를 벡터로 표현하는 것을 의미합니다. 워드 임베딩은 단어를 밀집 표현(Dense Representation)으로 변환하는 방법을 말합니다.

DTM 결과값을 보면 많은 부분이 0으로 채워져 있는 것을 위의 예제에서 확인할 수 있습니다. 이렇게 대부분의 요소가 0으로 채워지고 그 외의 값들만 저장하는 행렬을 희소 행렬(Sparse Matrix)라고 합니다.

밀집 표현은 희소 행렬의 반대되는 표현으로 벡터의 차원을 단어 집합의 크기로 지정하지 않습니다. 사용자가 설정한 값으로 모든 단어의 벡터의 차원을 동일하게 만듭니다. 이 과정에서 0과 1을 가진 아니라 실수값을 가지게 됩니다.

단어를 밀집 벡터의 형태로 표현하는 방법을 워드 임베딩(word embedding)이라고 하고, 임베딩 과정에서 나온 결과물을 임베딩 벡터(embedding vector)라고 합니다.

워드 임베딩 방법에는 Word2vec, FastText, Glove등이 있고, 여기서는 word2vec에 대해서 알아보고 pytorch로 실습하겠습니다.

Word2vecd은 구글의 토마스 미콜로프(Tomas Mikolov)가 이끄는 연구팀에서 2013년에 발표한 논문 “Efficient Estimation of Word Representations in Vector Space”에서 발표되었습니다. 해당 논문을 시작으로 단어 임베딩에 대한 관심이 뜨거워졌습니다.

word2vec은 딥러닝을 사용하여 만들었지만, 학습에 사용되는 모델 자체는 두개 계층만을 사용한 구조를 가지고 있습니다. 기존 단어 임베딩 기술과 다른점은 word2vec은 두 개의 계층만으로 사용해서 단순해지면서 계산양이 적어졌고 그로 인해서 더 많은 양의 데이터를 활용할 수 있다는 점입니다. 모델의 복잡도보다 데이터의 양이 더 중요하다는 점을 보여주었습니다.

논문에서 CBOW(Continuous Bag-Of-Words)와 Skip-gram을 제시하였습니다. 이 모델에서 단어의 의미는 문맥(context)에 의해 정의된다는 아이디어를 기반으로 합니다. 문맥은 주변 단어로 표현됩니다.

아래와 같이 친구가 당신에게 이야기 했다고 가정하겠습니다. 이렇게 들은 ㅁㅁ에 해당하는 글자가 처음 듣는 글자라고 해도 해당 ㅁㅁ가 음식이라고 추측할 수 있습니다.

“내일 영희를 만나는데 중국집에서 ㅁㅁ 먹자고 하네.”

이렇게 우리가 추측을 할 수 있는 이유는 주변 단어의 정보들이 추측하는데 도움이 되기 때문입니다.

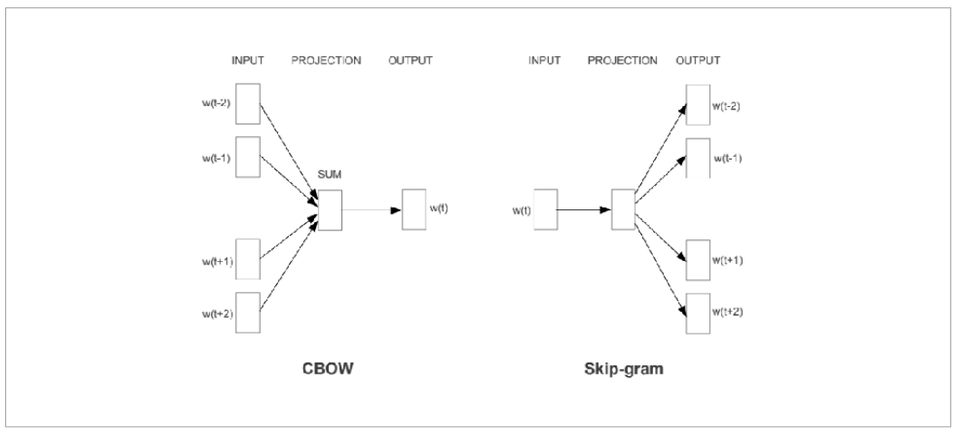
word2vec은 문맥은 현재 단어의 앞 단어와 뒤 단어 N개로 표현합니다. N은 하이퍼파라미터로 벡터화를 진행하는 사람이 직접 정해야 하는 값입니다. 논문에서 N의 값을 4~5를 사용했습니다.

위의 예제에서 ㅁㅁ 앞 뒤 2개의 단어를 사용하면 [만나는데, 중국집에서], [먹자고, 하네]가 해당됩니다. 이 값을 사용하여 의미를 파악합니다.

CBOW는 문맥 단어를 기반으로 현재 단어를 예측하는 모델입니다. 특정 단어를 중심으로 이전 n개의 단어와 이후 n개의 단어가 주어졌을 때, 중심 단어를 예측하는 것을 목표로 합니다.

Skip-Gram은 현재 단어를 기반으로 문맥 단어를 예측하는 모델입니다. 중심 단어가 주어졌을 때 이전 n개의 단어와 이후 n개의 단어를 예측합니다.

[그림 2-6] CBOW와 Skip-gram 모델



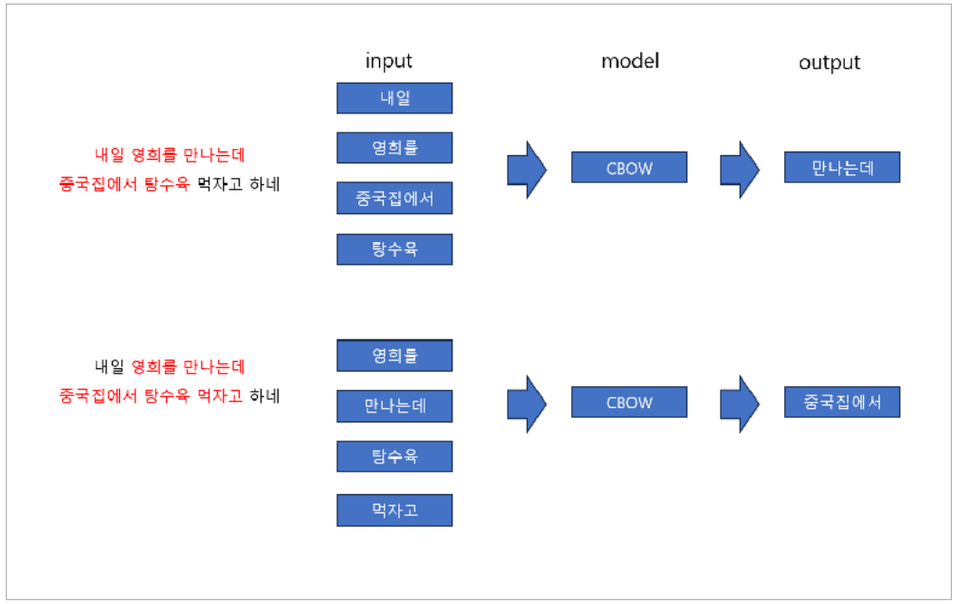
[만나는데, 중국집에서], [먹자고, 하네] 문맥을 활용해서 □□를 예측했다면 CBOW 방식이고, □□를 사용해서 [만나는데, 중국집에서], [먹자고, 하네]를 예측했다면 Skip-gram 방식입니다.

위의 예제에서 □□를 탕수육으로 변경하고 각각 모델이 어떻게 학습하는지 그림을 통해서 알아보겠습니다.

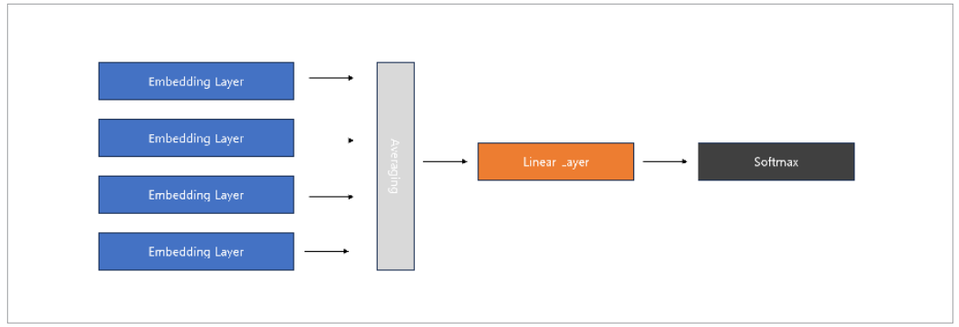
"내일 영희를 만나는데 중국집에서 탕수육 먹자고 하네"

그림 2-7은 CBOW 방식으로 n의 값을 2로 설정하여 순차적으로 학습하는 모습입니다. input으로 들어가는 실제 값은 해당 단어의 원핫(one-hot) 인코딩 된 벡터값이 입력됩니다. 출력층의 값도 해당 단어의 원핫 벡터의 값입니다.

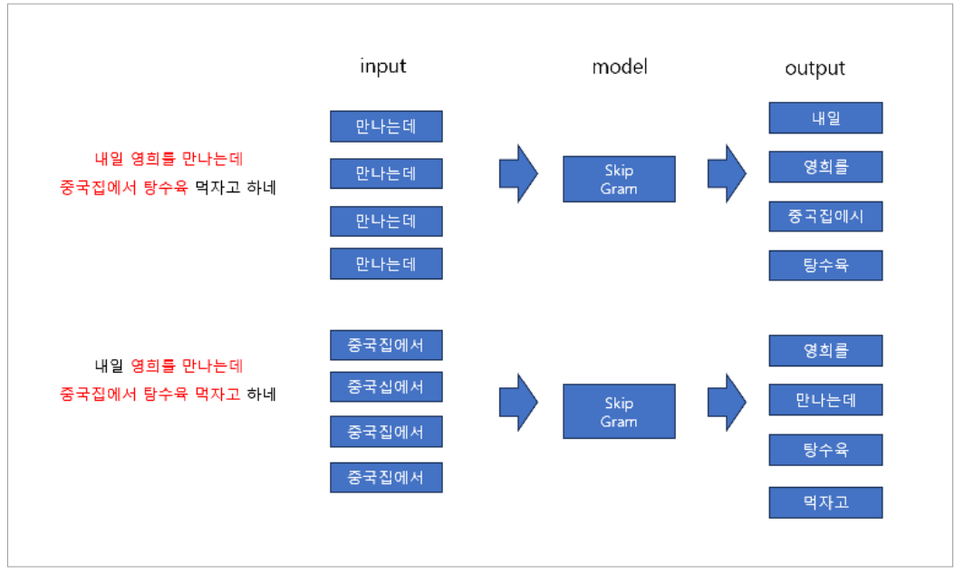
[그림 2-7] CBOW 학습 방법



[그림 2-8] CBOW Model Layer



[그림 2-9] Skip-Gram 학습 방법



[그림 2-10] Skip-Gram Model Layer

word2vec 모델의 입력값은 학습 말뭉치의 단어의 개수입니다. 말뭉치의 단어들을 원핫(One-Hot)으로 생성된 구조의 크기를 갖습니다.

논문에서 임베딩 레이어의 차원을 300으로 설정했을 때 가장 우수하다고 발표하였습니다. 임베딩 레이어에서 통과한 값은 선형 레이어를 거쳐 소프트맥스 활성화 함수를 통과해서 출력됩니다. 소프트맥스가 사용되는 이유는 단어의 수만큼 다중 분류 클래스 모델로 생성하기 때문에 최종 활성화 함수에 소프트맥스가 사용되었습니다.

CBOW, Skip-Gram의 차이점은 입력 단어의 수에 있습니다. CBOW 모델은 여러 단어를 입력 받아서 각각 동일한 임베딩 레이어를 거친 다음 단어 임베딩 벡터의 평균을 구하고 선형 레이어로 전달합니다. Skip-Gram은 단일 단어를 사용하기 때문에 평균을 구하지 않습니다.

논문의 저자는 말뭉치가 작을 때와 자주 사용하지 않은 단어들이 주어졌을 때는 Skip-Gram이 잘 작동한다고 강조했다. 같은 말뭉치라도 Skip-Gram 접근 방식이 훈련 샘플이 더 많이 얻을 수 있습니다. CBOW은 자주 쓰이는 단어들에 대해 정확도가 높고, 훈련 속도가 더 빠르게 동작합니다.

word2vec를 이용해서 노래 가사 100곡을 임베딩 벡터로 만들어 보겠습니다.

gemsim 외부 라이브러리를 사용해서 word2vec를 구현해보겠습니다. 해당 라이브러리는 따로 설치해야 합니다.

```
pip install gensim
```


제공된 노래 가사를 읽어서 OKt 형태소 분석를 사용하여 명사만 추출합니다. stopwords 리스트를 만들어 불용어를 처리하여 결과 품질을 높일 수 있습니다.

```
import os
from konlpy.tag import Okt
from gensim.models import Word2Vec

stopwords = ['처럼', '의','가','이','은','들','는','좀','잘','강','과','도','를','으로','자','에','와','한','하다',
            '알다', '₩n', '₩n₩n', '₩n₩n₩n']
text = ""
total = []
for file in os.listdir("./data"):
    with open("./data/{}".format(file), "r", encoding='utf8') as f:
        tokenized_sentence  = okt.nouns(f.read()) # 토큰화
        tokenized_word = [word.lower() for word in tokenized_sentence if not word in stopwords and len(word) > 1]
        total.append(tokenized_word)
```

gensim안에 word2vec이라는 함수를 사용하여 word2vec 모델을 생성하겠습니다. 해당 파라미터의 옵션값들은 아래와 같습니다.

vector_size = 워드 벡터의 특징 값. 즉, 임베딩 된 벡터의 차원.

window = 컨텍스트 윈도우 크기

min_count = 단어 최소 빈도 수 제한
(빈도가 적은 단어들은 학습하지 않음)

workers = 학습을 위한 프로세스 수

sg = 0은 CBOW, 1은 Skip-gram.

임베딩 차원을 100차원으로 CBOW형태로 생성해보겠습니다.

```
model = Word2Vec(sentences = total, vector_size = 100,
                 window = 5, min_count = 3, workers = 4, sg = 0
```

'사랑'이라는 단어와 유사도가 높은 단어들의 목록입니다.

```
print(model.wv.most_similar("사랑"))
```

```
[('디시', 0.8842909502983093), ('생각', 0.7881775498390198), ('이제', 0.7689843773841858), ('지금', 0.7639873027801514),
('보고', 0.760758936405181 9), ('메리', 0.7569893598556519), ('그대', 0.7560190558433533), ('시간', 0.7491391897201538),
('친구', 0.746405363082885 7), ('우리', 0.7452764511108398)]
```

Skip-Gram 방식으로 학습을 시킨 모델과 비교해보겠습니다.

```
model2 = Word2Vec(sentences = total, vector_size = 100,
                 window = 5, min_count = 3, workers = 4, sg = 1)
print(model2.wv.most_similar("사랑"))
```

```
[('이제', 0.9984060525894165), ('순간', 0.9983915090560913), ('우리', 0.9983357191085815), ('내게', 0.9982972741127014),
('메리', 0.9982879757881165), ('지금', 0.9982831478118896), ('모습', 0.9982772469520569), ('거리', 0.9982591271400452),
('가슴', 0.9982277750968933), ('설레', 0.9982190728187561)]
```

CBOW와 Skip-Gram이 출력한 결과가 다르다는 것을 확인할 수 있습니다.

위에서 결과의 수치 값은 두 벡터간의 코사인 유사도의 값입니다. 마지막으로 코사인 유사도에 대해서 알아보겠습니다.

코사인 유사도(cosine similarity)는 내적공간의 두 벡터간 각도의 코사인값을 이용하여 측정된 벡터간의 유사한 정도를 의미합니다. 각도가 0°일 때의 코사인값은 1이며, 다른 모든 각도의 코사인값은 1보다 작습니다.

따라서 이 값은 벡터의 크기가 아닌 방향의 유사도를 판단하는 목적으로 사용되며, 두 벡터의 방향이 완전히 같을 경우 1, 90°의 각을 이룰 경우 0, 180°로 완전히 반대 방향인 경우 -1의 값을 갖는다. 이 때 벡터의 크기는 값에 아무런 영향을 미치지 않는다. 코사인 유사도는 특히 결과값이 [0,1]의 범위로 떨어지는 수 공간에서 사용됩니다.

코사인 유사도는 어떤 개수의 차원에도 적용이 가능하여 흔히 다차원의 양수 공간에서의 유사도 측정에 자주 이용됩니다. 위의 예제처럼 단어 하나 하나는 각각의 차원을 구성하고 다차원 공간에서 코사인 유사도는 두 단어의 유사를 측정할 수 있습니다.

코사인 유사도가 널리 사용되는 이유 중 하나는 이것이 양수 공간이라는 조건만 만족하면 얼마나 많은 차원 공간이던지 거리를 측정하는 것이 가능하기 때문입니다.

[그림 2-11] 코사인 유사도 공식

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$