7. Basic Deep Learning Language Model

- Transformers
- word2vec

Sogang University Department of Business Administration
Professor Myung Suk Kim

Language model

◆ Language model basics

✓ A model that assigns probabilities to word sequences (arrangements of words).

✓ The probability of a whole sequence of words is equal to the chain of probabilities of the next word given the previous words. A language model is a model that assigns the probability of the next word appearing given the previous words

✓ P(a1, a2, a3) = P(a1) x P(a2|a1) x P(a3|a1, a2)

P(I, Sogang University, I am a student) =

P(I) x P(Sogang University| I) x P(I am a student|I, Sogang University)

◆ Language model evolution

✓ N-gram model: Calculate the probability for each word by the frequency of the preceding word

-> context is not considered

✓ Word2Vec-based model: Deep learning-based model trained with perceptron. Expressing word-to-word relationships -> no consideration of context

✓ RNN- based model : It has a recursive structure that processes each element of the sequence sequentially and uses previous information for the current calculation. As the length of the input sequence increases, it is difficult to maintain long-term dependency on information (long-term dependency problem). Modified RNN structures such as LSTM and GRU have emerged, but performance deteriorates as sentences become longer.

✓ Attention-based model: Attention vectors are generated and applied

✓ Transformer-based model: Self-attention is used.

-> Reason : (1) Operation per layer (2) Parallel processing

(3) Dependency learning between distant words

Ex) BERT, GPT

Language model

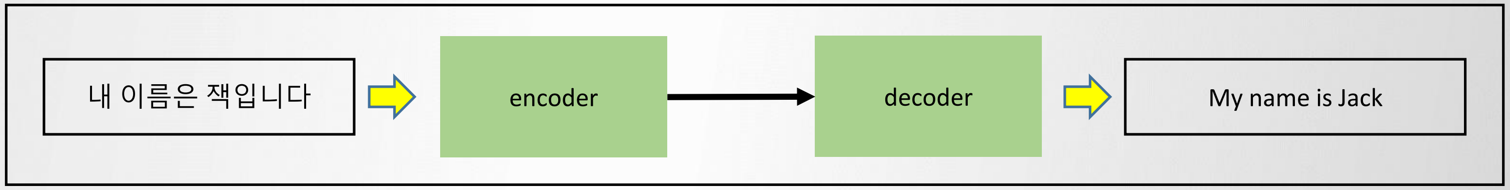
◆ Difference between RNN-based model and attention-based model

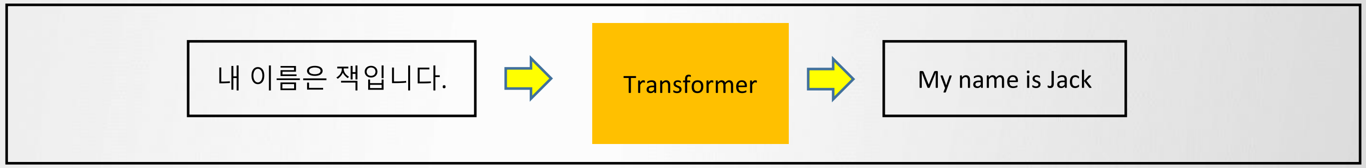
1. Sequence processing method:
   Attention: Processes all elements of the input sequence simultaneously and calculates the relevance between each element. Each element obtains the processing result through a weighted sum.
   RNN: Processes the sequence step by step, passing information from the previous step to the current step. Calculations are performed sequentially based on previous information, and the hidden state serves to maintain previous information.
2. Handling long-term dependencies:
   Attention: Calculates the relevance between each element within the input sequence to further highlight and focus important information.
   RNN: Difficulty handling long-term dependencies. Information loss and gradient vanishing problems occur in long sequences
3. Parallel processing:
   Attention: Advantageous for parallel processing because the relevance between each element can be calculated in parallel
   RNN: Difficult to parallelize because it is calculated sequentially, and processing speed is slow on large data
4. Sequence length handling:
   Attention: processed in the same way regardless of the length of the input sequence
   RNN: The disadvantage is that the longer the input sequence, the more difficult it is to maintain information.
5. Memory requirements:
   Attention: Memory requirements may be high because information within the input sequence is stored in the form of a weighted sum.
   RNN: Because it is computed sequentially, memory requirements can be relatively low.

Transformer

◆ Transformer Model Summary

- Transformer has a sequence-to-sequence model structure proposed by Google in 2017 and is widely used in machine translation.
- It is also widely used for positive and negative classification of review articles.
- Reduced learning time by not using LSTM or RNN
- In the case of a positive or negative decision, the word with strong attention applied is visualized to show the basis for the decision (explainable artificial intelligence)
- Possible to express a word vector depending on the context: Each word has various meanings, but it can have different meanings depending on the context
- An encoder-decoder model. When an input sentence (original text) is input to the encoder, the encoder learns how to express the input sentence and sends the result to the decoder. The decoder receives the expression result learned from the encoder and generates the desired sentence.

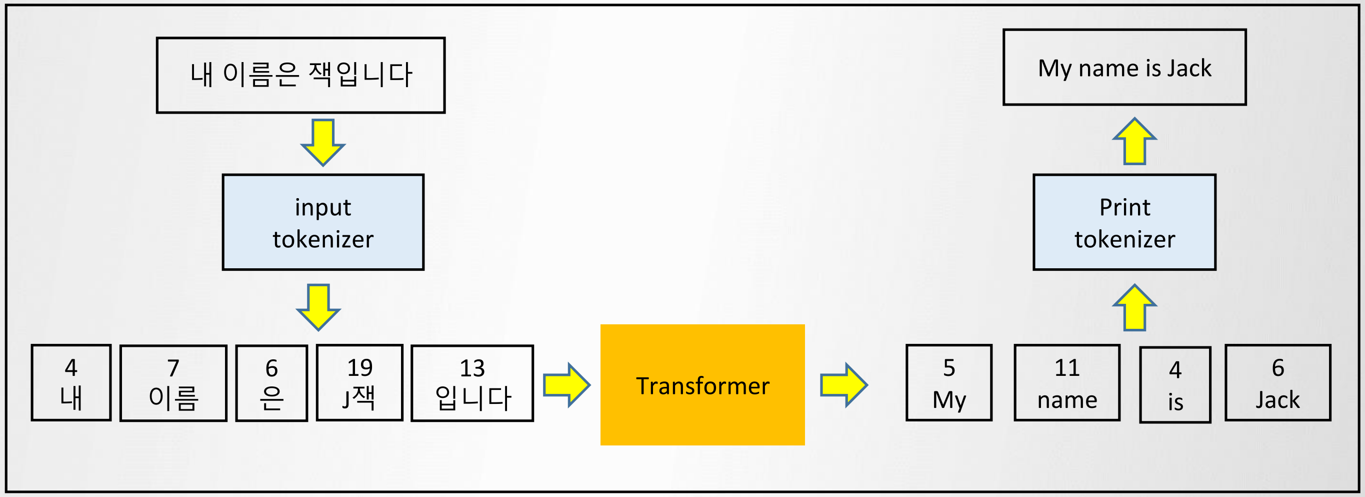




Transformer

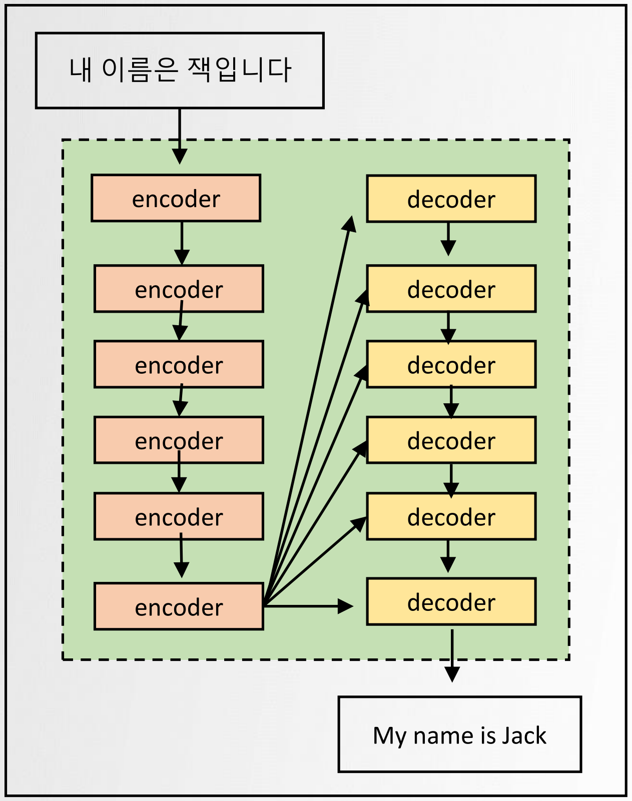
◆ Transformers model structure (e.g. machine translation)

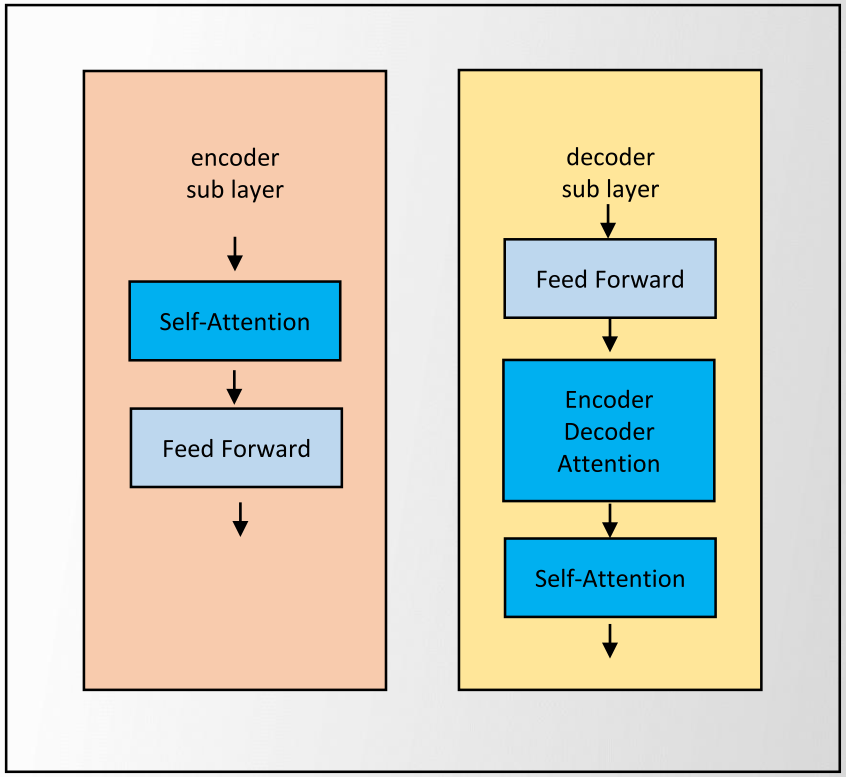
- After tokenizing with a tokenizer, each token is mapped to a number
- The tokenized input becomes the input of the transformer, and the transformer uses the input value to output a token and reverse the token to complete the sentence.



Transformer

- Encoder and decoder are composed of 6 sub-layers each
- It is connected in a row to the encoder sublayer and input values are passed in order, but the decoder sublayer receives the output of the previous sublayer and the last output value of the encoder as input.
- The sub-layer structure of the encoder and decoder is shown in the right figure.





Transformer

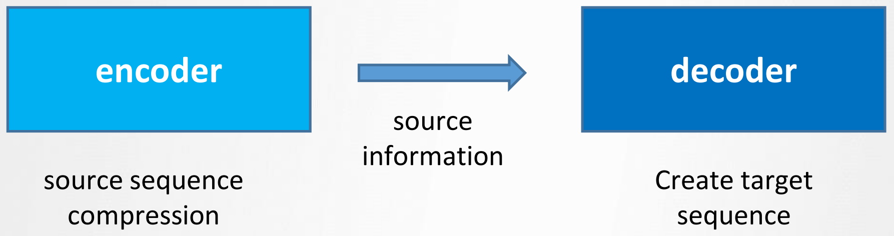
◆ Sequence to Sequence (Seq2Seq) Structure

- ✓ sequence : a sequence of words
- ✓ Sequence-to-Sequence : Converting a sequence with one property into a sequence with another property.
- ✓ ex) 어제, 에버랜드, 갔었어, 거기, 사람, 많더라 I, went, to, the, ever-land, there, were, many, people, there

sauce Language (6 Tokens) ⇨ Target Language (10 Tokens) _

◆ Encoder and decoder

- A model that performs a sequence-to-sequence task consists of an encoder and a decoder.
- The encoder is responsible for compressing the source sequence information and sending it to the decoder.
- Encoding: The process of compressing source sequence information
- Decoding: The process of generating a target sequence by receiving information sent by the encoder



Transformer

◆ Model training and inference

- ✓ the encoder and decoder inputs are given.
- ✓ Ex) Encoder : I went to Everland yesterday there were many people

(Step 1) Decoder input : <s> → target Create Sequence : I

(Step 2) Decoder input : <s> I → target Create Sequence : went

(Step 3) Decoder input : <s> I went → target Create Sequence : to

(Reference) <s> is a special token meaning start , </s> is a special token meaning end

◆ Transformer block

- ✓ Encoder block : A repetitive element among the encoders [Multi-Head Attention], [Feed Forward neural network], [residual connection and Layer Normalization] consists of three elements
- ✓ Decoder block : Similar to the encoder block , but [Mask multi-head attention], [Multi-head using the input of head attention]

◆ Transformer-based pre-training model

- ✓ GPT: A forward learning model that has strengths in sentence generation and uses only a decoder.
- ✓ BERT: A two-way learning model that has strengths in extracting the meaning of sentences using a mask language model and uses only an encoder.

Transformer

◆ Attention

- ✓ Attention is primarily a mechanism used to improve the interaction of encoders and decoders.
- ✓ It is mainly used in sequence-to-sequence models, and is used to model relationships between different sequences by learning how each element of an input sequence contributes to which element of an output sequence.
- ✓ As a type of machine learning performed on sequence input, it is a technique to improve performance by extracting information centered on meaningful elements.
- ✓ RNN-based sequence-to-sequence models work by encoding an input sequence into a fixed-length vector and then generating an output sequence through a decoder. At this time, since the length of the input sequence and the output sequence may be different, it may be difficult to learn the relationship between sequences of different lengths.
- ✓ To solve this problem, the Attention mechanism was introduced. Attention refers to a mechanism that allows a decoder to "focus" on a different part of an input sequence for each output location. This allows the decoder to generate an output by dynamically considering different parts of the input sequence.
- ✓ Attention mechanisms can be used with RNNs, but attention itself is not dependent on RNNs or any other specific model. Recently, attention plays an important role in models such as the Transformer architecture, and these models can achieve very good natural language processing results without RNN.

Ex) When decoding a target language in machine translation, translation quality is improved by selecting words that are helpful for decoding among word sequences in the source language.

Transformer

◆ Attention calculation example

✓ Suppose we are given input vectors A and B.

A = [1, 2, 3], B = [4, 5, 6]

Calculate the dot product between vectors A and B:

dot_product = A[0] * B[0] + A[1] * B[1] + A[2] * B[2] = 1 * 4 + 2 * 5 + 3 * 6 = 32

Afterwards, apply the softmax function to convert the dot product value to probability.

attention_weight = softmax(dot_product)

✓ Let's assume we are given an input sentence and a query vector.

Assume that each word in a sentence is represented by a vector

Sentence: "I love natural language processing" Query: "natural"

Calculate the similarity (Dot Product) between each word vector and the query vector:

Similarity = [dot("natural", "I"), dot("natural", "love"), dot("natural", "natural"), dot("natural", "language"), dot("natural", "processing")]

Convert the similarity value to probability using the softmax function:

attention_weight = softmax(similarity)

✓ Modeling the relationship between two vectors, mainly used to determine the relationship between different sequences

Transformer

◆ Self attention

✓ [Multi-head attention] is also called [self-attention]

✓ A mechanism for modeling relationships between different elements within an input sequence, mainly used in models such as Transformer

✓ Learning how each element of an input sequence relates to each other

✓ Identify dependencies and relationships between words in a sentence, and use them to extract features or model context

◆ Self-attention calculation method

✓ Structure in which the three elements of query, key, and value influence each other

✓ Query: A vector representing a specific word vector and used to calculate similarity through the inner product operation with other word vectors.

✓ Key: A word vector used when calculating similarity. Similarity is calculated through the inner product operation between Query and Key.

✓ Value: A word vector used when calculating the final attention weight, which is multiplied by the attention weight to form the output.

✓ Words within a sentence are entered in vector form in the transformer block.

✓ Each word vector is converted into a query, key, and value through an intra-block calculation process. The number of queries, keys, and values matches the number of words in the sentence.

✓ When self-attention calculation for all sequences is completed, the result is passed to the next block and repeated as many times as the number of blocks (layers).

Transformer

◆ Self-attention (Scaled Dot-Product Attention) calculation example

✓ Assume we are given a single vector

X = [0.1, 0.2, 0.3, 0.4]

Calculate the dot product between vector X and itself:

dot_product = X[0] * X[0] + X[1] * X[1] + X[2] * X[2] + 0.3

Scale: Divide the dot product by the square root of d_model (d_model is the number of vector dimensions)

scaled_dot_product = dot_product / sqrt(d_model)

Afterwards, the softmax function is applied to convert the dot product value into probability.

attention_weight = softmax(scaled_dot_product)

✓ Represent the entire input sentence as one vector and calculate self-attention

Sentence: "I love natural language processing"

Calculate the similarity (Dot Product) between each word vector in a sentence and itself:

Similarity = [dot("I", "I"), dot("love", "love"), dot("natural", "natural"), dot("language", "language"),
dot("processing", "processing")]

Scale the similarity value and convert it to probability using the softmax function:

attention_weight = softmax(similarity)

✓ Modeling relationships between different elements within a vector, mainly between words within a sequence

It is used to determine dependencies and context

Transformer

- ◆ Self-Attention calculation using Query, Key, and Value methods
- ✓ Sentence: “The cat is sitting on the mat“

Query, Key, Value assignment:

Query: "sitting“

Key: "The", "cat", "is", "sitting", "on", "the", "mat"

Value: Each word vector (omitted)

Dot product calculation between each Key and Query:

dot product("sitting", "The") = ...
dot product("sitting", "cat") = ...
dot product("sitting", "is") = ...
dot product("sitting", "sitting") = ...
dot product("sitting", "on") = ...
dot product("sitting", "the") = ...
dot product("sitting", "mat") = ...

Similarity calculation using dot product:

similarity("sitting", "The") = softmax(dot product("sitting", "The"))
similarity("sitting", "cat") = softmax(dot product("sitting", "cat"))
similarity("sitting", "is") = softmax(dot product("sitting", "is"))
similarity("sitting", "sitting") = softmax(dot product("sitting", "sitting"))
similarity("sitting", "on") = softmax(dot product("sitting", "on"))
similarity("sitting", "the") = softmax(dot product("sitting", "the"))
similarity("sitting", "mat") = softmax(dot product("sitting", "mat"))

This similarity value is used as the attention weight.

Transformer

◆ Self -attention and other deep learning techniques

✓ Convolution Neural Network (CNN): A model that captures the local characteristics of a sequence using a convolutional filter. It plays an important role in forming the meaning of the surrounding context based on a specific word, but it is difficult to read the context beyond the size of the convolutional filter.

예) 어제 에버랜드 갔는데, 거기 사람 많더라
어제 에버랜드 갔는데, 거기 사람 많더라
어제 에버랜드 갔는데, 거기 사람 많더라
어제 에버랜드 갔는데, 거기 사람 많더라

I went to Everland yesterday. There were many people
I went to Everland yesterday. There were many people
I went to Everland yesterday. There were many people
I went to Everland yesterday. There were many people

✓ Recurrent Neural Network (RNN): The source sequence is processed in turn, and the longer the sequence length, the more information compression problems occur, and the higher the recent information weight, the more the meaning of the final word ('people') is reflected.

Ex) I -> went -> to -> Everland -> yesterday -> There -> were -> many -> people

✓ Attention : Consider the context as a whole by calculating attention on individual words and entire sequences (CNNs are local)

Unaffected by sequence length (RNNs forget preceding words)

Attention is performed once when connecting only all words in the source sequence (yesterday, Everland,..., there were many) and one word in the target sequence (Everland) and generating one target word.

✓ Self Attention: Attention performed on itself, connecting all words in the source sequence with all words in the input sequence. It works without RNN and iterates as many times as the number of encoder and decoder blocks when generating one target word.

Transformer

◆ Classification of language models according to learning direction

✓ Forward language model : trains a pre-trained model from front to back (GPT)

예) 어제 에버랜드 ( )　　I went to Everland ( ).

어제 에버랜드 갔는데, 거기 사람 ( )　　I went to Everland yesterday. There were many ( ).

✓ Inverse language model : trains a pre-trained model from the back of a sentence to the front

예) ( ) 에버랜드 갔는데, 거기 사람 많더라　　( ) went to Everland yesterday. There were many people

( ) 갔는데, 거기 사람 많더라　　( ) were many people

✓ Mask Language Model : Learning by creating blanks in the learning target and classifying what would be an appropriate word (BERT)

예) 어제 에버랜드 갔는데, 거기 ( ) 많더라　　I went to Everland ( ). There were many people

어제 에버랜드 갔는데, ( ) 사람 많더라　　I went to Everland yesterday. There were ( ) people

✓ 스킵-그램 모델: Learning by setting a specific range before and after a word and classifying which words will come within this range (Word2vec)

예) 어제 에버랜드 갔는데, ( ) 사람 많더라　　I went to Everland yesterday. ( ) were many people

어제 에버랜드 ( ), 거기 사람 많더라　　I went to Everland ( ). There were many people

Word2vec

◆ Word2vec (Tomasi Mikolov , MS Apprentice , 2012)

- A topic vector has meaning as a word if it is only in the same sentence, but word2vec has meaning as a word nearby.
- Using n-grams consisting of words around the target word
- Specify Window=k to specify k words to include before and after that word
- Find word2vec mainly using back propagation of neural network structure.
- It uses the probability of surrounding words and corresponds to unsupervised learning.

✓ Type :

- If a word vector suitable for use in a specific area is required, word2vec training is required with a text corpus related to the area.
- CBOW (Continuous Bag of Words) and Skip-Gram

✓ Pros :

- Since text data is based on order, LSTM or RNN training is common, but it takes a long time to learn.
- Word2vec is not using LSTM or RNN among deep learning techniques.

✓ Cons :

- Word2vec is a static embedding technique, so multiple meanings corresponding to a single word are fixedly expressed as only one vector, and homophones cannot be distinguished because they have the same expression value wherever they appear in the document text (regardless of order).
- As a solution to this, contextual embedding techniques such as BERT and ELMo, which generate dynamic vectors for each context based on sentences, have been developed.
- By modifying the embedding layer of the pretraining model, BERT can have a vector expression value optimized for a specific area through appropriate transfer learning.

Word2vec

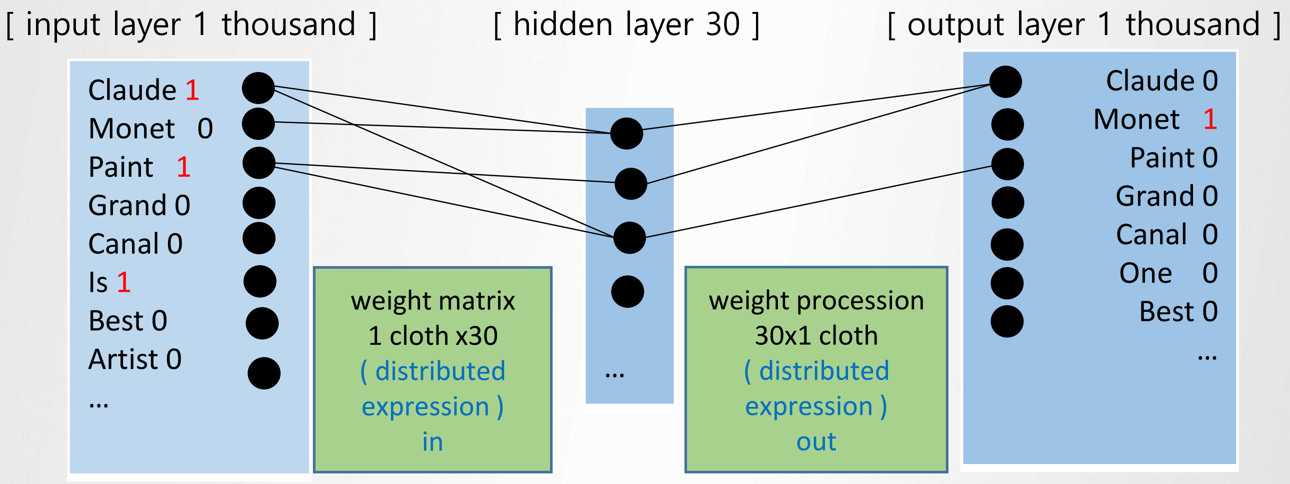
(1) CBOW

- How to predict a word through surrounding words in context

<Procedure>

1. Make a one-hot vector of each neighboring word and use it as the value of the input layer (vocabulary vector: mx1) Specify window=k and code only n words in the front and rear as 1 and the rest as 0
2. Each one-hot vector is multiplied by a weight matrix (variance representation: nxk) to create a k-dimensional hidden layer vector (topic vector: kx1)
   (learn weight matrix with deep learning model)
3. Multiply the k-dimensional vector by the weight matrix (variance expression: kxn) again to obtain a vector of the same dimension as the one-hot vector.
   Created (output layer vector: mx1)
4. Find the word with 1 in the output layer vector.

Ex) Document : { Claude Monet painted the Grand Canal. Claude Monet is the best artist in...
If window=1, it is used to find the word that will be underlined in {Claude ______ painted... } .



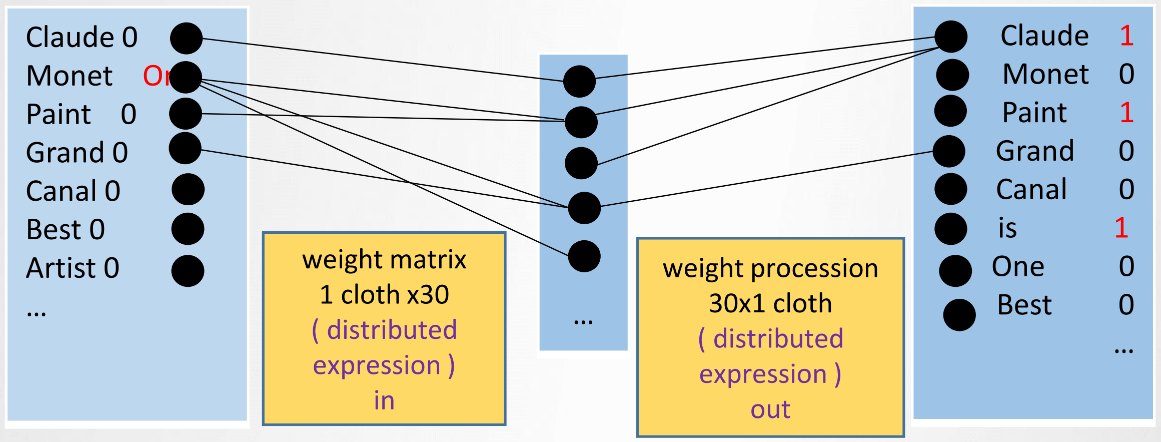
Word2vec

(2) Skip-Gram

- A method of predicting surrounding words in a specific context with a certain word
- Changed CBOW and input/output.

Ex) Document : { Claude Monet painted the Grand Canal. Claude Monet is the best artist in...
with window=1, it is used to find the words that will be underlined in { ______ Monet ________ } .

[input layer 1 thousand] [hidden layer 30] [output layer 1 thousand]



Word2vec

◆ Comparison of CBOW and Skip-Gram

- Skip-Gram is useful when a corpus is small and infrequently used words are given, and CBOW is known to have high accuracy and fast training speed for frequently used words.
- There is no theoretical proof of which word vector representation is better to use between CBOW and Skip-Gram, but since Skip-Gram expresses only a specific word as 1 in the input layer, the hidden layer can represent only that word.
  The word vector representation can be said to be easy to reflect the characteristics of the word in terms of the Skip-Gram method in terms of wanting a vector representation of a single word.

Ex) In case of Skip-Gram

A weight matrix with 3 neurons (usually using the in weight matrix)

Monet Word Circle in 6 Word Vocabulary One-Hot Vector

3D word vectors (word embedding)

Claude	Monet	painted	the	Grand	Canal
0	One	0	0	0	0

X

.03	.92	.66
.06	.32	.71
.14	.62	.43
.24	.99	.62
.12	.02	.44
.32	.23	.55

=

.06
.32
.71

⇦ Word vector expressing Monet

Document vector creation : using Word2vec/ Doc2vec

◆ Using Word2vec (https://wikidocs.net/102705)

- After obtaining the word vectors, the average of the word vectors existing in the document is regarded as the document vector.

Ex) Document 1: { Claude Monet painted the Grand Canal}

< word vector (word2vec value) present in document 1 > < vector in document 1 : mean word vector >

Claude	Monet	painted	the	Grand	Canal
.03	.06	.14	.24	.12	.32
.92	.32	.62	.99	.02	.23
.66	.71	.43	.62	.44	.55

Doc1
.1517
.5167
.5683

In this way, vectors of all documents can be obtained, and similarity between documents and cluster classification can be performed using them.

◆ Using

- Doc2Vec is an algorithm that transforms Word2Vec to obtain the embedding of a document.
- Like Word2Vec, implemented through Gensim, a Python machine learning package

Paper title : Distributed Representations of Sentences and Documents
Paper link : https://arxiv.org/abs/1405.4053

Word2vec

```
####### Word2Vec Basic1 #######
# https://m.blog.naver.com/PostView.nhn?isHttpsRedirect=true&blogId=wideeyed&logNo=221349385092&categoryNo
=49&proxyReferer=

from gensim.models import Word2Vec
import matplotlib.pyplot as plt

sentence_ex = [
                ['this', 'is', 'a',   'good',       'product'],
                ['it',    'is', 'a',   'excellent', 'product'],
                ['it',    'is', 'a',   'bad',        'product'],
                ['that', 'is', 'the', 'worst',      'product']
            ]


# Create words and vectors using sentences
model1 = Word2Vec(sentence_ex, vector_size=300, window=3, min_count=1, workers=1)
# Hyperparameter value
# size ( or word_size )= feature value of word vector . i.e. , the dimension of the embedded vector .
# window = context window size ( up to 3 words before and after )
# min_count = limit on the minimum frequency of words ( words with low frequency are not learned .)
# workers = number of processes for training
# sg = 0 is CBOW, 1 is Skip-gram (sg= 1 is default )

type(model1) # gensim.models.word2vec.Word2Vec
```

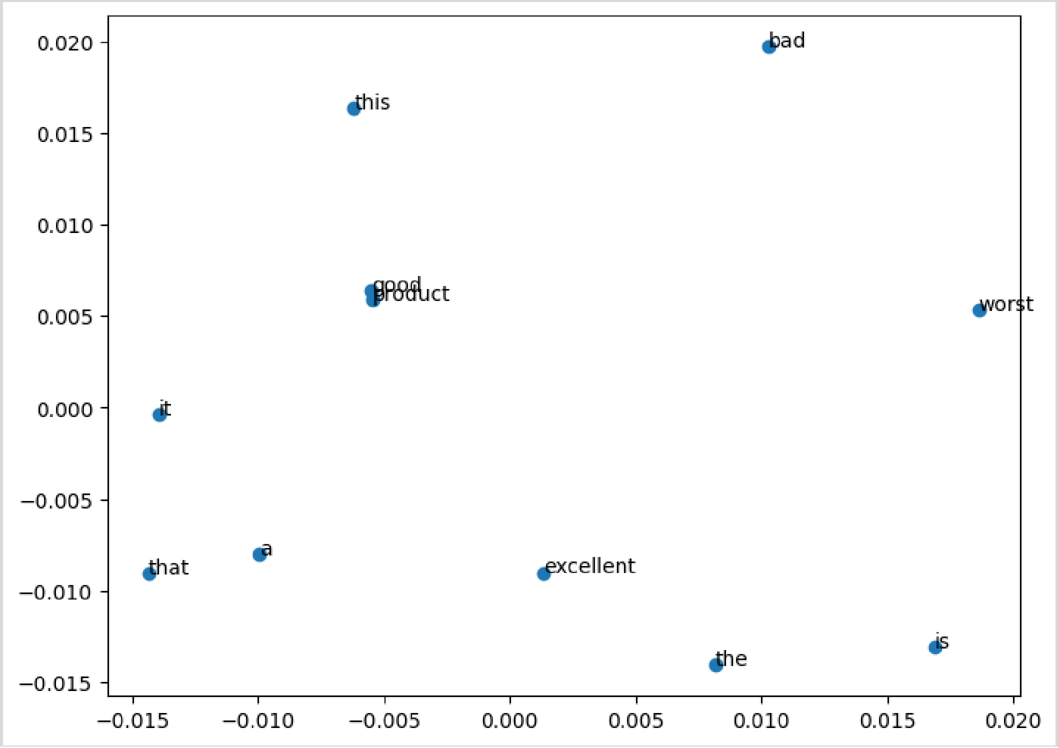
Word2vec

```
# get word vectors
word_vectors1 = model1.wv
vocabs1 = word_vectors1.index_to_key
print(vocabs1)
len(vocabs1)  # 11 words
type(vocabs1)

word_vectors_list1 = [word_vectors1[v] for v in vocabs1]
list(vocabs1)[0]
word_vectors1['this']
len(word_vectors1['this’])  # 300
type(word_vectors1) # gensim.models.keyedvectors.KeyedVectors

len(word_vectors_list1) # vectors for 11 words
word_vectors_list1[0]
len(word_vectors_list1[0]) # 300 vector elements for each word
print(word_vectors_list1[0])


# similarity between words
print(word_vectors1.similarity(w1='it', w2='this'))
word_vectors1.most_similar("it")
# find the most similar word ( using cosine similarity )
```

```
[('product', 0.0813601091504097),
('that', 0.07037851214408875),
('this', 0.06809870898723602),
('a', 0.0500403456389904),
('the', 0.018759682774543762),
('excellent', 0.005325536709278822),
('good', -0.026010606437921524),
('worst', -0.03475712612271309),
('bad', -0.04481656849384308),
('is', -0.08273911476135254)]
```

Word2vec

```
# similarity between words Two- dimensional visualization representation
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
xys = pca.fit_transform(word_vectors_list1)
xs = xys[:,0]
ys = xys[:,1]

# Draw a 2D graph by inputting the word and the 2D X- axis value and Y -axis value
def plot_2d_graph(vocabs, xs, ys):
    plt.figure(figsize=(8 ,6))
    plt.scatter(xs, ys, marker = 'o')
    for i, v in enumerate(vocabs):
        plt.annotate(v, xy=(xs[i], ys[i]))
plot_2d_graph(vocabs1, xs, ys)
```



Word2vec

```
####### Pre-trained model application #######
# Pre-trained model: can increase accuracy
import gensim.downloader
# Show all available models in gensim-data
print(list(gensim.downloader.info()['models'].keys()))

# Download the "glove-twitter-25" embeddings
glove_vectors = gensim.downloader.load('glove-twitter-25')
# glove_vectors = gensim.downloader.load('word2vec-google-news-300')

# Use the downloaded vectors as usual:
glove_vectors.most_similar('twitter')
type(glove_vectors)  # gensim.models.keyedvectors.KeyedVectors


# Check similarity between words
print(glove_vectors.similarity(w1='it', w2='this'))

# Similar words(rank)
print(glove_vectors.most_similar("it"))
```

Word2vec

```
####### Word2Vec Basic2 #######
# https://towardsdatascience.com/a-beginners-guide-to-word-embedding-with-gensim-word2vec-model-5970fa56cc92
# Car data download https://www.Kaggle.com/datasets/CooperUnion/cardataset

!pip install --upgrade genism
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib notebook
import numpy as np
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
import re  # For preprocessing
import pandas as pd  # For data handling
from time import time  # To time our operations
from collections import defaultdict  # For word frequency

import spacy  # For preprocessing
from gensim.models import Word2Vec
import logging  # Setting up the loggings to monitor gensim
logging.basicConfig(format="%(levelname)s - %(asctime)s: %(message)s", datefmt= '%H:%M:%S', level=logging.INFO)
from sklearn.manifold import TSNE
from numpy import dot
from numpy.linalg import norm
from google.colab import drive
drive.mount('/content/drive')
```

Word2vec

```
df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Textmining/download/car.csv')
df.head()

# (1) Data Preprocessing
# Create a new column for Make Model
df['Maker_Model']= df['Make']+ " " + df['Model']
df.head()

# Select features from original dataset to form a new dataframe
df1 = df[['Engine Fuel Type','Transmission Type','Driven_Wheels','Market Category','Vehicle Size', 'Vehicle Style', 
'Maker_Model’]]

# For each row, combine all the columns into one column
df2 = df1.apply(lambda x: ','.join(x.astype(str)), axis=1)

# Store them in a pandas dataframe
df_clean = pd.DataFrame({'clean': df2})

# Create the list of list format of the custom corpus for gensim modeling
sent = [row.split(',') for row in df_clean['clean’]]

# show the example of list of list format of the custom corpus for gensim modeling
sent[:2]
```

Word2vec

```
# (2) Genism word2vec Model Training
model = Word2Vec(sent, min_count=1, vector_size= 50,workers=3, window =3, sg = 1)
model.wv['Toyota Camry’]


# (3) Compute Similarities
model.wv.similarity('Porsche 718 Cayman', 'Nissan Van’)

model.wv.similarity('Porsche 718 Cayman', 'Mercedes-Benz SLK-Class’)

model.wv.most_similar('Mercedes-Benz SLK-Class')[:5]
```

Word2vec

```
# cosine-distance 이용
def cosine_distance (model, word,target_list , num) :
    cosine_dict ={}
    word_list = []
    a = model[word]
    for item in target_list :
        if item != word :
            b = model [item]
            cos_sim = dot(a, b)/(norm(a)*norm(b))
            cosine_dict[item] = cos_sim
    dist_sort=sorted(cosine_dict.items(), key=lambda dist: dist[1],reverse = True) ## in Descedning order
    for item in dist_sort:
        word_list.append((item[0], item[1]))
    return word_list[0:num]

# only get the unique Maker_Model
Maker_Model = list(df.Maker_Model.unique())

# Show the most similar Mercedes-Benz SLK-Class by cosine distance
cosine_distance (model.wv,'Mercedes-Benz SLK-Class',Maker_Model,5)
```

Word2vec application

1. English word2vec (TED XML data: Two reasons companies fail-and how to avoid them)

```
#https://wikidocs.net/50739
(1) Data download
import nltk
nltk.download('punkt')
import urllib.request
import zipfile
from lxml import etree
import re
from nltk.tokenize import word_tokenize, sent_tokenize
from google.colab import drive
drive.mount('/content/gdrive')

# download
url = "https://raw.githubusercontent.com/GaoleMeng/RNN-and-FFNN-textClassification/master/ted_en-20160408.xml"
file_name = "/content/gdrive/My Drive/Colab Notebooks/Textmining/download/ted_en-20160408.xml"
urllib.request.urlretrieve(url, filename=file_name)
```

Word2vec application

(2) Preprocess: remove xml grammar

```
targetXML=open(file_name, 'r', encoding='UTF8')
target_text = etree.parse(targetXML)

# From xml file, bring the contents between <content> and </content>
parse_text = '₩n'.join(target_text.xpath('//content/text()'))

# Using sub module in regular expression, remove (Audio), (Laughter), etc. in the contents
# Delete the contents composed of bracket
content_text = re.sub(r'₩([^)]*₩)', '', parse_text)

# Tokenization using NLTK in corpus
sent_text = sent_tokenize(content_text)

# Remove punctuations. Transform fromm big letters into small letters
normalized_text = []
for string in sent_text:
     tokens = re.sub(r"[^a-z0-9]+", " ", string.lower())
     normalized_text.append(tokens)

# Tokenization using NLTK for each sentence
result = [word_tokenize(sentence) for sentence in normalized_text]

print('총 샘플의 개수 : {}'.format(len(result)))

# extract 3 samples
for line in result[:3]:
    print(line)
```

Word2vec application

(3) word2vec 훈련 및 실행

```
# Hyperparameter value
# size ( or word_size )= feature value of word vector . i.e. , the dimension of the embedded vector .
# window = context window size
# min_count = limit on the minimum frequency of words ( words with low frequency are not learned .)
# workers = number of processes for training
# sg = 0 is CBOW, 1 is Skip-gram

from gensim.models import Word2Vec
model = Word2Vec(sentences=result, vector_size=100, window=5, min_count=5, workers=4, sg=0)
type(model)   # gensim.models.word2vec.Word2Vec

# model.wv.most_similar: most similar words are extracted
model_result = model.wv.most_similar("man")
print(model_result)
```

(4) word2vec model save and load

```
from gensim.models import KeyedVectors
model.wv.save_word2vec_format('eng_w2v') # model save
loaded_model = KeyedVectors.load_word2vec_format("eng_w2v") # model load
type(loaded_model)    # gensim.models.keyedvectors.KeyedVectors

# Extract similar words with man
loaded_model.similarity(w1='man', w2='lady')
model_result = loaded_model.most_similar("man")
print(model_result)
```

Word2vec application

2. Korean word2vec (NAVER Movie Review)

(1) Data download

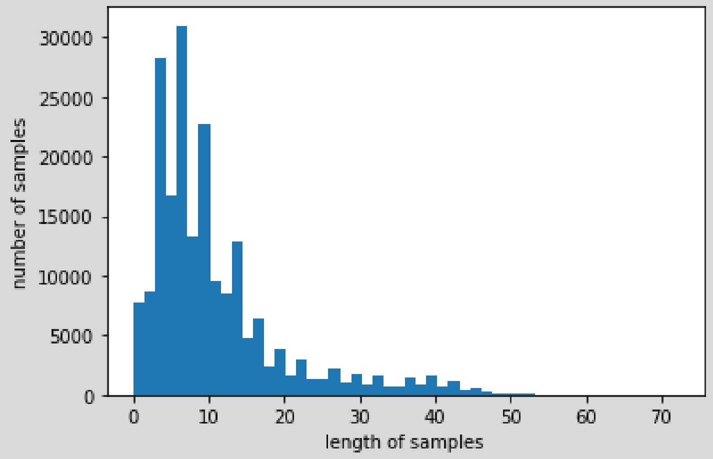
```
import pandas as pd
import matplotlib.pyplot as plt
import urllib.request
from gensim.models.word2vec import Word2Vec
from konlpy.tag import Okt
from google.colab import drive
drive.mount('/content/gdrive')

url1 = "https://raw.githubusercontent.com/e9t/nsmc/master/ratings.txt"
file_name1 = "/content/gdrive/My Drive/Colab Notebooks/Textmining/download/ratings.txt"
urllib.request.urlretrieve(url1, filename= file_name1)

# Data call
train_data = pd.read_table(file_name1)
train_data[:5]
print(len(train_data))
```

Word2vec application

(2) Pre-processing

```
print(train_data.isnull().values.any()) # NULL exist?
train_data = train_data.dropna(how = 'any') # remove rows with Null value
print(train_data.isnull().values.any()) # check if Null exist
print(len(train_data)) # number of review

# Remove other words than Korean using regular expression
train_data['document'] = train_data['document'].str.replace("[^ㄱ-ㅎㅏ-ㅣ가-힣 ]","")
train_data[:5] # top 5 words

# stop words
stopwords = ['의','가','이','은','들','는','좀','잘','걍','과','도','를','으로','자','에','와','한','하다']

# Tokenization using OKT (It takes time)
okt = Okt()
tokenized_data = []
for sentence in train_data['document']:
    temp_X = okt.morphs(sentence, stem=True) # 토큰화
    temp_X = [word for word in temp_X if not word in stopwords] # 불용어 제거
    tokenized_data.append(temp_X)

# Review length check
print('리뷰의 최대 길이 :',max(len(l) for l in tokenized_data))
print('리뷰의 평균 길이 :',sum(map(len, tokenized_data))/len(tokenized_data))
plt.hist([len(s) for s in tokenized_data], bins=50)
plt.xlabel('length of samples')
plt.ylabel('number of samples')
plt.show()
```

Word2vec application



(3) word2vec training and execution

```
from gensim.models import Word2Vec
model = Word2Vec(sentences = tokenized_data, vector_size = 100, window = 5, min_count = 5, workers = 4, sg = 0)

# check embedding matrix size
model.wv.vectors.shape
print(model.wv.most_similar("최민식"))
print(model.wv.most_similar("히어로"))
```

Document vector application : Word2vec

1. Book Recommendation System (using wod2vec average)

```
# https://wikidocs.net/102705
import urllib.request
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import requests
import re
from PIL import Image
from io import BytesIO
from nltk.tokenize import RegexpTokenizer
import nltk
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
from nltk.corpus import stopwords
from sklearn.metrics.pairwise import cosine_similarity
import nltk
nltk.download('stopwords’)

# document download
urllib.request.urlretrieve("https://raw.githubusercontent.com/ukairia777/tensorflow-nlp-
tutorial/main/09.%20Word%20Embedding/dataset/data.csv", filename="data.csv")
df = pd.read_csv("data.csv")
print('전체 문서의 수 :',len(df))

df[:10]  # top 10 print
```

Document vector application : Word2vec

```
# Preprocessing
def _removeNonAscii(s):
    return "".join(i for i in s if  ord(i)<128)

def make_lower_case(text):
    return text.lower()

def remove_stop_words(text):
    text = text.split()
    stops = set(stopwords.words("english"))
    text = [w for w in text if not w in stops]
    text = " ".join(text)
    return text

def remove_html(text):
    html_pattern = re.compile('<.*?>')
    return html_pattern.sub(r'', text)

def remove_punctuation(text):
    tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
    text = tokenizer.tokenize(text)
    text = " ".join(text)
    return text
```

Document vector application : Word2vec

```
df['cleaned'] = df['Desc'].apply(_removeNonAscii)
df['cleaned'] = df.cleaned.apply(make_lower_case)
df['cleaned'] = df.cleaned.apply(remove_stop_words)
df['cleaned'] = df.cleaned.apply(remove_punctuation)
df['cleaned'] = df.cleaned.apply(remove_html)

df['cleaned'][:10]  # After preprocessing, print top 10

# If there is empty row, transform into nan, then delete the row
df['cleaned'].replace('', np.nan, inplace=True)
df = df[df['cleaned'].notna()]
print('전체 문서의 수 :',len(df))

# Tokenization -> save corpus(list type) -> train Word2Vec
corpus = []
for words in df['cleaned']:
    corpus.append(words.split())

# (1) Model building using the given dataset
word2vec_model = Word2Vec(corpus, vector_size = 300, window=3, min_count = 2, workers = -1)
type(word2vec_model)   # gensim.models.word2vec.Word2Vec
```

Document vector application : Word2vec

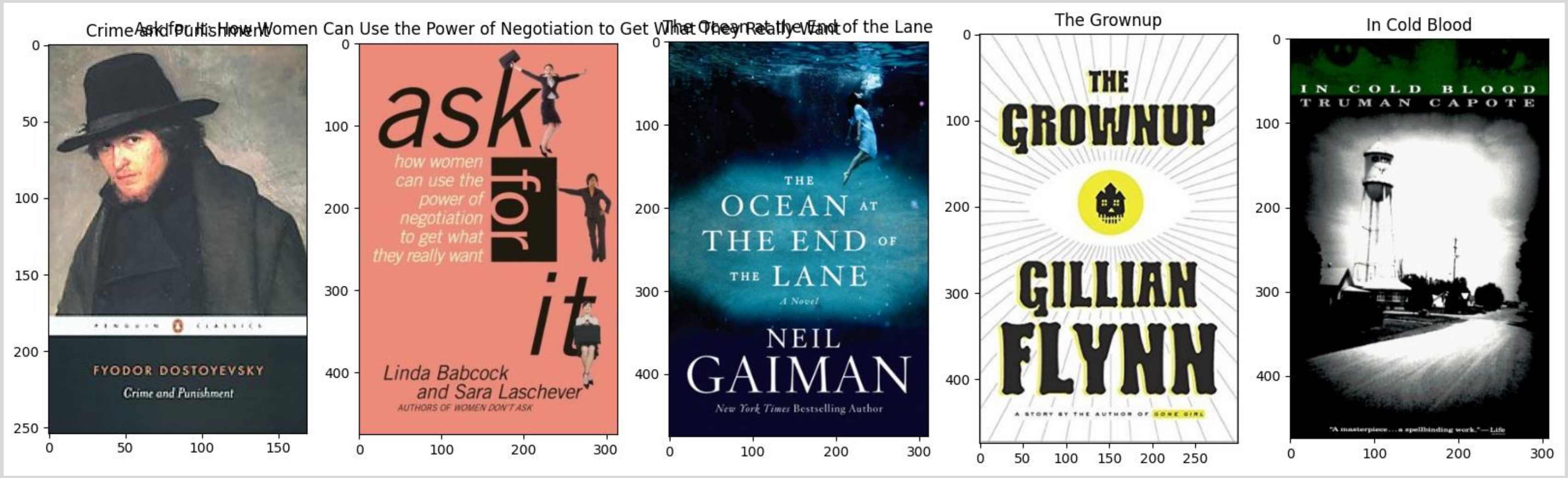
```
# compute the average of word vector
def get_document_vectors(document_list):
    document_embedding_list = []

    # For each document
    for line in document_list:
        doc2vec = None
        count = 0
        for word in line.split():
            if word in word2vec_model.wv.index_to_key:
                count += 1
                # add word vector values of all documents
                if doc2vec is None:
                    doc2vec = word2vec_model.wv[word]
                else:
                    doc2vec = doc2vec + word2vec_model.wv[word]

        if doc2vec is not None:
            # the vector of sum of word vector is divided by length of vector
            doc2vec = doc2vec / count
            document_embedding_list.append(doc2vec)

    # Return document vector list for each document
    return document_embedding_list
document_embedding_list = get_document_vectors(df['cleaned'])
print('문서 벡터의 수 :',len(document_embedding_list))  # 2381
```

Document vector application : Word2vec

```
# Recommendation system
# Cosine similarity between documents
from sklearn.metrics.pairwise import cosine_similarity
cosine_similarities = cosine_similarity(document_embedding_list, document_embedding_list)
print('코사인 유사도 매트릭스의 크기 :',cosine_similarities.shape)

# Find 5 books with the most similar plot using cosine similarity for the selected book
def recommendations(title):
    books = df[['title', 'image_link']]

    # If you enter the title of a book, the index of the title is returned. Save to idx
    indices = pd.Series(df.index, index = df['title']).drop_duplicates()
    idx = indices[title]

    # Select 5 books with a similar plot (document embedding) to the entered book
    sim_scores = list(enumerate(cosine_similarities[idx]))
    sim_scores = sorted(sim_scores, key = lambda x: x[1], reverse = True)
    sim_scores = sim_scores[1:6]

    # Index of 5 most similar books
    book_indices = [i[0] for i in sim_scores]

    # Extract only the row at that index from the entire data frame. It has 5 rows
    recommend = books.iloc[book_indices].reset_index(drop=True)

    fig = plt.figure(figsize=(20, 30))
```

Document vector application : Word2vec

```
    # Sequentially print images from dataframe
    for index, row in recommend.iterrows():
        response = requests.get(row['image_link'])
        img = Image.open(BytesIO(response.content))
        fig.add_subplot(1, 5, index + 1)
        plt.imshow(img)
        plt.title(row['title’])

recommendations("The Da Vinci Code")
```

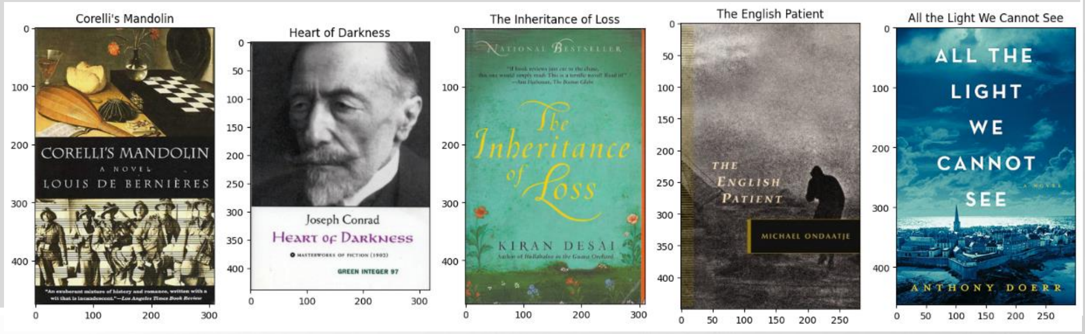

Crime and Punishment
Ask for It: How Women Can Use the Power of Negotiation to Get What They Really Want
The Ocean at the End of the Lane
The Grownup
In Cold Blood

Document vector application : Word2vec

```
# (2) Model building using pre-trained model
import gensim.downloader
glove_vectors = gensim.downloader.load('glove-twitter-25')
word2vec_model = glove_vectors
type(word2vec_model)

# Compute average of word vector
def get_document_vectors(document_list):
    document_embedding_list = []
    # 각 문서에 대해서
    for line in document_list:
        doc2vec = None
        count = 0
        for word in line.split():
            if word in word2vec_model.index_to_key:
                count += 1
                if doc2vec is None:
                    doc2vec = word2vec_model[word]
                else:
                    doc2vec = doc2vec + word2vec_model[word]
        if doc2vec is not None:
            doc2vec = doc2vec / count
            document_embedding_list.append(doc2vec)
    return document_embedding_list
document_embedding_list = get_document_vectors(df['cleaned'])
print('문서 벡터의 수 :',len(document_embedding_list))  # 2381
```

Document vector application : Word2vec

```
# Cosine similarity between documents
from sklearn.metrics.pairwise import cosine_similarity
cosine_similarities = cosine_similarity(document_embedding_list, document_embedding_list)

# Select 5 books with a similar plot (document embedding) to the entered book
def recommendations(title):
    books = df[['title', 'image_link']]
    indices = pd.Series(df.index, index = df['title']).drop_duplicates()
    idx = indices[title]
    sim_scores = list(enumerate(cosine_similarities[idx]))
    sim_scores = sorted(sim_scores, key = lambda x: x[1], reverse = True)
    sim_scores = sim_scores[1:6]
    book_indices = [i[0] for i in sim_scores]
    recommend = books.iloc[book_indices].reset_index(drop=True)
    fig = plt.figure(figsize=(20, 30))

 # Sequentially print images from dataframe
    for index, row in recommend.iterrows():
        response = requests.get(row['image_link'])
        img = Image.open(BytesIO(response.content))
        fig.add_subplot(1, 5, index + 1)
        plt.imshow(img)
        plt.title(row['title’])

recommendations("The Da Vinci Code")
```



문서 벡터 응용: doc2vec

1. Doc2Vect 기초

```
# https://swlock.blogspot.com/2020/01/doc2vec.html#google_vignette
#Import all the dependencies
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize

data = ["I love machine learning. Its awesome.",
        "I love coding in python",
        "I love building chatbots",
        "they chat amagingly well"]

tagged_data = [TaggedDocument(words=word_tokenize(_d.lower()), tags=[str(i)]) for i, _d in enumerate(data)]
print(tagged_data)


# 또는
tagged_data = []

for i, _d in enumerate(data):
 tagged_data.append(TaggedDocument(words=word_tokenize(_d.lower()), tags=[str(i)]))
tagged_data
```

문서 벡터 응용: doc2vec

```
max_epochs = 100
vec_size = 20
alpha = 0.025
model = Doc2Vec(vector_size=vec_size,
                alpha=alpha,
                min_alpha=0.00025,
                min_count=1,
                dm =1)
model.build_vocab(tagged_data)



model.iter = 50
for epoch in range(max_epochs):
    print('iteration {0}'.format(epoch))
    model.train(tagged_data,
                total_examples=model.corpus_count,
                epochs=model.iter)
    # decrease the learning rate
    model.alpha -= 0.0002
    # fix the learning rate, no decay
    model.min_alpha = model.alpha
model.save("d2v.model")
print("Model Saved")
```

문서 벡터 응용: doc2vec

```
from gensim.models.doc2vec import Doc2Vec

model= Doc2Vec.load("d2v.model")
#to find the vector of a document which is not in training data
test_data = word_tokenize("I love chatbots".lower())
v1 = model.infer_vector(test_data)
print("V1_infer", v1)


# to find most similar doc using tags
similar_doc = model.docvecs.most_similar('1')
print(similar_doc)
```

Document vector application: Doc2vec

2. Doc2Vect Basic (Korean)

```
# https://sosoeasy.tistory.com/326
!pip install konlpy
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from konlpy.tag import Okt

#Data preprocess
okt=Okt()
data = ["나는 학교에 간다",
        "나는 친구를 만난다",
        "나는 영화보러 간다",
        "영화가 재밌다",
        "나는 공부를 하러 왔다",
        "나는 영화관을 갔다"]

# Training should be taggedDocument type. So, do the following preprocessing steps
tagged_data = [TaggedDocument(words=okt.morphs(_d), tags=[str(i)]) for i, _d in enumerate(data)]
for i in tagged_data:
    print(i)
```

Document vector application: Doc2vec

```
# Model building
max_epochs = 100
model = Doc2Vec(vector_size=20,
                alpha=0.025,
                min_alpha=0.00025,
                min_count=1,
                dm=1)
model.build_vocab(tagged_data)
type(model)

# Training
model.iter = 150
for epoch in range(max_epochs):
    model.train(tagged_data,
                total_examples=model.corpus_count,
                epochs=model.iter)    # decrease the learning rate
    model.alpha -= 0.0002             # fix the learning rate, no decay
    model.min_alpha = model.alpha

type(model)
sample="영화가 보고싶군".split()
print(model.infer_vector(sample))

# Model test
similar_doc = model.dv.most_similar(model.infer_vector(sample))
print(similar_doc)
```

Document vector application: Doc2vec

3. Similarity computing between Corona NAVER news using Doc2Vect

```
#https://wikidocs.net/155356

# Install Mecab
!pip install konlpy
!pip install mecab-python
!bash <(curl -s https://raw.githubusercontent.com/konlpy/konlpy/master/scripts/mecab.sh)

import pandas as pd
from konlpy.tag import Mecab
from gensim.models.doc2vec import TaggedDocument
from tqdm import tqdm

# Document load
from google.colab import drive
drive.mount('/content/gdrive')

df = pd.read_csv('/content/gdrive/MyDrive/Colab Notebooks/Textmining/download/코로나_naver_news1.csv',  sep=',')
df = df.dropna()
Df
Df['title'][0]
```

Document vector application: Doc2vec

```
# To learn Doc2Vec, two things are required: the 'title' of the document and the 'body' of the document in word tokenized state
# Stores the 'title' of the document in tags of TaggedDocument and the result of word tokenization corresponding to the 'body' of the
# document in words, Create a tagged_corpus_list, a Python list with elements of this result
mecab = Mecab()

tagged_corpus_list = []

for index, row in tqdm(df.iterrows(), total=len(df)):
  text = row['description']
  tag = row['title']
  tagged_corpus_list.append(TaggedDocument(tags=[tag], words=mecab.morphs(text)))
print('문서의 수 :', len(tagged_corpus_list))

tagged_corpus_list[0]  # check the result of document preprocessing of the first document

# Doc2Vec train and test
from gensim.models import doc2vec
model = doc2vec.Doc2Vec(vector_size=300, alpha=0.025, min_alpha=0.025, workers=8, window=8)

# Vocabulary build
model.build_vocab(tagged_corpus_list)
#print(f"Tag Size: {len(model.dv.doctags.keys())}", end=' / ')

# Doc2Vec train
model.train(tagged_corpus_list, total_examples=model.corpus_count, epochs=50)

# Model save
model.save('/content/gdrive/My Drive/Colab Notebooks/Textmining/download/corona_naver_news.doc2vec')

# Model test
similar_doc = model.dv.most_similar(df['title'][0])
print(df['title'][0])
print(similar_doc)
```