Assignments
Applications of Natural Language Processing (Intro to NLP and its recent progress)
Fall 2022

Graduate School of Data Science
Seoul National University
Instructor: Dr.Jay-yoon Lee (lee.jayyoon@snu.ac.kr)
Head TA: Jonghyun Song (hyeongoon11@snu.ac.kr)

Due on Thursday Sep. 29, 2022 by 3:30pm (before class)

(Assignment 1: Word and Sentence Representation)

Honor Pledge for Graded Assignments

"I, HEEJIN, CHAE , affirm that I have not given or received any unauthorized help on this assignment, and that this work is my own."

0 Instructions (4)

- Total score cannot exceed 100 points. For example, if you score 95 points from non-bonus questions and 10 points are added from bonus questions, your score will be 100 points, not 105 points.
- You can download the skeleton code file HERE
- You can download the TeX file HERE
- Skeleton codes for problem 1, 2, 3, and 4 are at the directory `/skipgram`, `/fasttext`, `/sentence`, and `/tfidf` each.
- Run the `bash collect_submission.sh` script to produce your 2022_abcde_coding.zip file. Please make sure to modify collect_sumbssion.sh file before running this command. (**abcde** stands for your student id)
- Modify this tex file into `2022_abcde_written.pdf` with your written solutions
- Upload both `2022_abcde_coding.zip` and ://www.overleaf.com/project/632c396c9d73cc88b3d14167exttt2022_abcd to etl website. **(4pts)**

1 Understanding Skip-Gram (36)

1.1 Sofmax Loss Function

Word representation by **Word2vec** is theoretically supported by *distributional hypothesis*[1] In the example below, a 'center' word *banking* is surrounded by 'outside' words *turning, into, crises*, and *as* when the context window length is 2.

The objective of training Skip-gram is to learn the conditioned probability distribution of outside word O given center word C, $P(O = o|C = c)$. (i.e. the probability of the word o is an 'outside' word for word c) The probability can be obtained by taking the softmax function over the inner product of word vectors as below:

$$P(O = o \mid C = c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{w \in \text{Vocab}} \exp(\boldsymbol{u}_w^\top \boldsymbol{v}_c)} \tag{1}$$

where u_o is the 'outside vector' representing outside word o, and v_c is the 'center vector' representing center word c. Be aware that outside and center vector representations of the same word are defined differently.

[1] The hypothesis that words that occur in similar contexts have similar meanings.

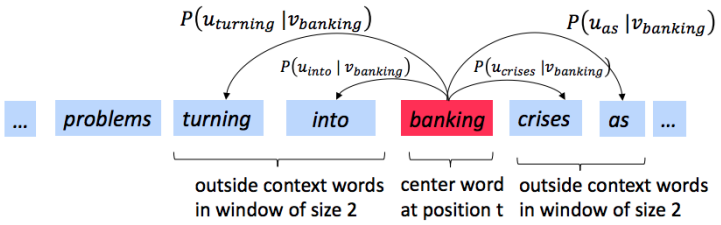


Figure 1: The skip-gram prediction model with window size 2

For the whole vocabulary, we can store outside vector u_w and center vector v_w in two matrices, U and V by columns. That is, the columns of U are all the outside vectors u_w and the columns of V are all the center vectors v_w for every $w \in Vocabulary$

The loss for a single pair of words c and o is defined as:

$$J_{\text{naive-softmax}}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log P(O=o|C=c) = -\log \frac{\exp(u_o^\top v_c)}{\sum_{w\in \text{Vocab}} \exp(u_w^\top v_c)} = -u_o^T v_c + \log \sum_{w\in vocab} \exp(u_w^T v_c) \tag{2}$$

To learn the correct center and outside vector by gradient descent, we need to compute the partial derivative of the outside and center word vectors. Partial derivative of $J_{\text{naive-softmax}}$ with respect to v_c can be obtained as following:

$$\frac{\partial J}{\partial v_c} = -u_o^T + \frac{\sum_{i\in vocab} u_i^T \exp(u_i^T v_c)}{\sum_{w\in vocab} \exp(u_w^T v_c)} = -u_o^T + \sum_{i\in vocab} P(O=i|C=c)u_i^T = -u_o^T + \sum_{w\in vocab} \hat{y}_w u_w^T = -\mathbf{y}^T U^T + \hat{\mathbf{y}}^T U^T \tag{3}$$

where $\mathbf{y}$ is the one-hot vector which has 1 at the index of true outside word o ($y_o = 1$ and 0 for all other $i != o$, $y_i = 0$), and 0 everywhere else while $\hat{\mathbf{y}}$ stands for $P(O|C=c)$, a prediction made by Skip-gram model in equation (1).

(a) When is the gradient (3) is zero? **(4pts)** **Answer:** 예측값과 실제 라벨(label)이 같을 때 ($\hat{\mathbf{y}} = \mathbf{y}$)

(b) If we update v_c with this gradient, toward which vector is v_c getting pulled to? **(4pts)**

- **HINT**: The loss for a single pair of words c and o can be thought of as the cross-entropy between the true distribution y and the predicted distribution $\hat{y}$ for a particular center word c and outside word o.

$$\boldsymbol{J}_{\text{naive-softmax}}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log P(O=o|C=c) = -\sum_{w\in vocab} y_w \log(\hat{y}_w) = -\log \hat{y}_o. \tag{4}$$

Answer: 주어진 실제 outside word인 O의 백터인 u_o

(c) Derive the partial derivatives of $J_{\text{naive-softmax}}$ with respect to the outside word vector matrix, U.

- **Note 1**: To get full credit, please write your answer in the form of $v_c, \mathbf{y}$, and $\hat{\mathbf{y}}$ and write the whole computation process. That is, the correct answer does not refer to other vectors but $v_c, \mathbf{y}$, and $\hat{\mathbf{y}}$.
- **Note 2**: Be careful about the dimensions of arrays and matrices. The final answer should have the same shape as U. (i.e. (vector length) $\times$ (vocabulary size))
- **HINT**: Start from outside word vector u_w, which is the column vector of U. Then, dividing into two cases may help: when $w = o$ and $w \neq o$ **(6pts)**

Answer: when $w \neq o$

$\frac{\partial J}{\partial u_w} = 0 + \frac{exp(u_w^T v_c)}{\sum_{w\in vocab} \exp(u_w^T v_c)} v_c = P(O=w|C=c)v_c = \hat{\mathbf{y}} v_c$

when $w = o$

$\frac{\partial J}{\partial u_w} = -v_c + \frac{exp(u_w^T v_c)}{\sum_{w\in vocab} \exp(u_w^T v_c)} v_c = -v_c + P(O=w|C=c)v_c = (\mathbf{y} - \hat{\mathbf{y}})v_c$

1.2 Negative Sampling Loss

Now we shall consider the Negative Sampling loss, which is an alternative to the Naive Softmax loss. Assume that K negative samples (words) are randomly drawn from the vocabulary. For simplicity of notation we shall refer to them as $w_1, w_2, \ldots, w_K$, and their outside vectors as $\boldsymbol{u}_{w_1}, \boldsymbol{u}_{w_2}, \ldots, \boldsymbol{u}_{w_K}$. For a center word c and an outside word o, the negative sampling loss function is given by:

$$\boldsymbol{J}_{\text{neg-sample}}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log\big(\sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c)\big) - \sum_{s=1}^{K} \log\big(\sigma(-\boldsymbol{u}_{w_s}^\top \boldsymbol{v}_c)\big) \tag{5}$$

for a sample $w_1, \ldots w_K$, where $\sigma(\cdot)$ is the sigmoid function.

(a) Compare the computational complexity of negative sampling with that of the softmax loss function presented in section 1.1. State the reason why negative sampling is computationally more efficient. **(6pts)**

Answer:
$O(\boldsymbol{J}_{\text{neg-sample}}) = O(-\log\big(\sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c)\big) - \sum_{s=1}^{K} \log\big(\sigma(-\boldsymbol{u}_{w_s}^\top \boldsymbol{v}_c)\big)) = O(1+K) = O(K)$
$O(\boldsymbol{J}_{\text{naive-softmax}}) = O(u_o^T v_c + \log \sum_{w \in vocab} \exp\big(u_w^T v_c\big)) = O(1+V) = O(V)$

Where
K =size of Negative samples
V = size of All vocabularies
K < V

$\therefore O(\boldsymbol{J}_{\text{neg-sample}}) < O(\boldsymbol{J}_{\text{naive-softmax}})$ Negative-sampling의 $\sum$ 부분에서 K개의 샘플에대해서만 계산하고, Naive-softmax는 $\sum$부분에서 모든 Vocabularies에 대해서 계산하기 때문에 Negative sampling이 더 효율적이다

(b) Compute the partial derivatives of $\boldsymbol{J}_{\text{neg-sample}}$ with respect to $\boldsymbol{v}_c$, $\boldsymbol{u}_o$, and the s^{th} negative sample $\boldsymbol{u}_{w_s}$. Please write your answers in terms of the vectors $\boldsymbol{v}_c$, $\boldsymbol{u}_o$, and $\boldsymbol{u}_{w_s}$, where $s \in [1, K]$. (within five lines) **(6pts) Answer:**
1) $\frac{\partial J}{\partial v_c} = -(1-\sigma(u_o^T v_c))u_o + \sum_{s=1}^{K}(1-\sigma(-u_{w_s}^T v_c))u_{w_s}$
2) $\frac{\partial J}{\partial u_o} = -(1-\sigma(u_o^T v_c))v_c$
3) $\frac{\partial J}{\partial u_{w_s}} = -\sum_{s=1}^{K}(\sigma(-u_{w_s}^T v_c) - 1)v_c$

1.3 Coding

In this part, you will implement the Skip-Gram model and train word vectors with stochastic gradient descent (SGD). Before you begin, first run the following commands within the assignment directory in order to create the appropriate conda virtual environment. This guarantees that you have all the necessary packages to complete the assignment. You will be asked to implement the math functions above using the Numpy package at the root directory.

```
conda env create -f env.yml
conda activate a1
```

Once you are done with the assignment you can deactivate this environment by running:

```
conda deactivate
```

For each of the methods you need to implement, we included approximately how many lines of code our solution has in the code comments. These numbers are included to guide you. You don't have to stick to them, you can write shorter or longer code as you wish. If you think your implementation is significantly longer than ours, it is a signal that there are some `numpy` methods you could utilize to make your code both shorter and faster. `for` loops in Python take a long time to complete when used over large arrays, so we expect you to utilize `numpy` methods. The sanity checking function is provided to check if your code works well.
To run the code command below, please move your directory to `/skipgram`.

(a) We will start by implementing methods in `word2vec.py`. You can test a particular method by running `python word2vec.py m` where `m` is the method you would like to test. For example, you can test the naiveSoftmaxLossAndGradient method by running `python word2vec.py naiveSoftmaxLossAndGradient`. (Copy and paste it to your terminal)

(i) Implement the softmax loss and gradient in the `naiveSoftmaxLossAndGradient` method. **(3pts)**

(ii) Implement the negative sampling loss and gradient in the `negSamplingLossAndGradient` method. **(3pts)**

(b) When you are done, test your entire implementation by running `python word2vec.py`. Now we are going to load some real data and train word vectors with everything you just implemented! We are going to use The Recognizing Textual Entailment (RTE) dataset to train word vectors. (You can check out the raw datasets in the directory `./skipgram/utils/datasets/RTE`. There is no additional code to write for this part; just run `python run.py`.

(**Note**: The training process may take a long time depending on the efficiency of your implementation and the computing power of your machine **(an efficient implementation takes around three to four hours)**. Please start your homework as soon as possible!)

After 40,000 iterations, the script will finish and visualization for your word vectors will appear. It will also be saved as `word_vectors.png` in your project directory. **Include the plot below (4pts)**

Answer:

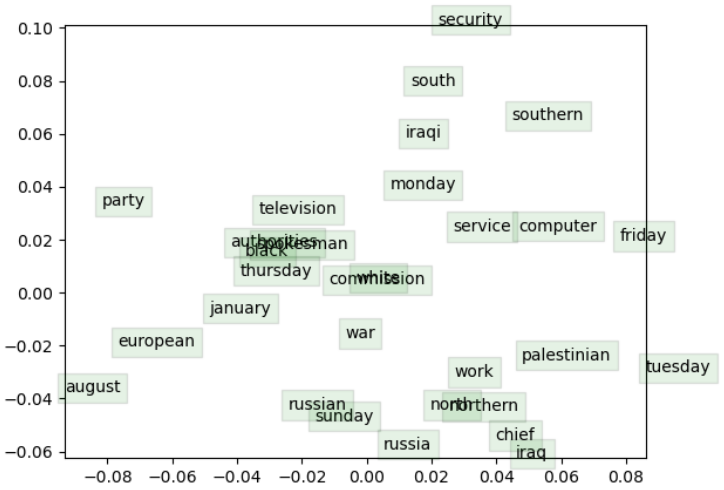


Figure 2: word2vec visualization

2 FastText (28+4)

2.1 FastText Loss Function

Let's move on to the FastText model. Similar to the Skip-gram model, the loss function we are trying to optimize (for a single context word) is given as follows:

$$\boldsymbol{J}(v_c, u_o, u_{w_s}) = -\log(\sigma(s(v_c, u_o))) - \sum_{s=1}^{K} \log(\sigma(-s(v_c, u_{w_s})))$$

$$s(w, u_o) = \sum_{g \in G_w} z_g^T u_o$$

We associate a vector representation z_g to each character n-gram g. Make sure you understand how this expression is written using the sigmoid (σ) is similar to the one in the paper. Extending this to every outside

word, we get the objective as:

$$\boldsymbol{J}_{\text{FastText}}(w_c, w_{t-m}, w_{t-m+1} \dots w_{t+m}) = \sum_{\substack{-m \le j \le m \\ j \neq 0}} \boldsymbol{J}(w_t, w_c, N_K)$$

The embeddings of the n-grams from the center words are from the *center* embeddings V while the embeddings for the outside words are drawn from *outside word* embeddings U.

(a) You train a Skip-gram model and a fastText model (with parameters as described in the paper) on a corpus of text. You train these models to have embeddings of size 128. Your corpus has 40000 unique words. Your model is upper-bounded by $2e6$ subword tokens.
How many parameters are there in the Skip-gram model and FastText model? **(4pts)**
For more details, please refer to the papers:

- **Skip-Gram**: https://arxiv.org/pdf/1310.4546.pdf
- **FastText**: https://arxiv.org/pdf/1607.04606.pdf

Answer:

(b) Calculate $\frac{\partial \boldsymbol{J}}{\partial u_o}$ and $\frac{\partial \boldsymbol{J}}{\partial u_{w_s}}$ **(6pts)**
Answer:
$\frac{\partial \boldsymbol{J}}{\partial u_o} = -\frac{\partial \log(\sigma(s(v_c, u_o)))}{\partial u_o} - \frac{\partial \sum_{s=1}^{K} \log(\sigma(-s(v_c, u_{w_s})))}{\partial u_o} = (\sigma(s(v_c, u_o)) - 1) \sum_{g \in G_{v_c}} z_g$
$\frac{\partial \boldsymbol{J}}{\partial u_{w_s}} = -\frac{\partial \log(\sigma(s(v_c, u_o)))}{\partial u_{w_s}} - \frac{\partial \sum_{s=1}^{K} \log(\sigma(-s(v_c, u_{w_s})))}{\partial u_{w_s}} = \sum_{s=1}^{K} (1 - (\sigma(-s(v_c, u_{w_s})))) \sum_{g \in G_{v_c}} z_g^T u_{w_s}$

(c) Calculate $\frac{\partial \boldsymbol{J}}{\partial z_g}$ **(4pts) Answer:** $(\sigma(s(v_c, u_o)) - 1) \sum_{g \in G_{v_c}} u_o + \sum_{s=1}^{K} (1 - \sigma(-s(v_c, u_o))) \sum_{g \in G_{v_c} u_o}$

2.2 Coding

(a) We will start by implementing methods in `./fasttext` directory. After setting your python(or conda) environment as instructed on Problem 1. You can test a particular method by running `python word2vec.py m` where `m` is the method you would like to test. For example, you can test the negSamplingLossAndGradient method by running `python fasttext.py negSamplingLossAndGradient`. (Copy and paste it to your terminal)
To run the code command below, please move your directory to `/fasttext`.

(i) Implement the negative sampling loss and gradient in the `negSamplingLossAndGradient` method in the `word2vec.py` of `./fasttext` directory. (**HINT**: Just copy and paste the code from the previous question. Be aware that the names of some arguments have changed.) **(3pts)**

(ii) Implement `generate_ngrams` function in the `utils/treebank.py` file. Sanity check by `python utils/treebank.py generate_ngrams`. **(3pts)**

(b) When you are done, run the entire implementation by command `python run.py`. (check out you are in the right directory `./fasttext`)

(**Note**: The training process may take a long time depending on the efficiency of your implementation and the computing power of your machine **(an efficient implementation takes around two hours)**. Please start your homework as soon as possible!)

After 20,000 iterations (which is shorter than word2vec), the script will finish and visualization for your word vectors will appear. It will also be saved as `word_vectors.png` in your project directory. **Include the plot below** How is the plot different from the one generated earlier from Skip-Gram? **(4pts)**

Answer: 같은 subword를 가지는 단어들에 대해서 성능이 높아진것을 볼 수 있음(day 등)

(c) What do you think are the advantages of a subword approach such as fastText compared to word2vec? (state within one line) **(4pts)**
Answer: 학습속도가 훨씬 빠름.

(d) **(Bonus)** Can you think of a few examples where subword approach might hurt the embedding's performance? (two or three examples are enough) **(4pts)**
Answer: 1) 의미는 다르지만 subword가 같은 단어가 유사한 단어로 처리될 수 있음(spokesman, russian)
2) 한 단어를 여러번 sampling하는 효과로 bias가 더 짙어질 수 있음

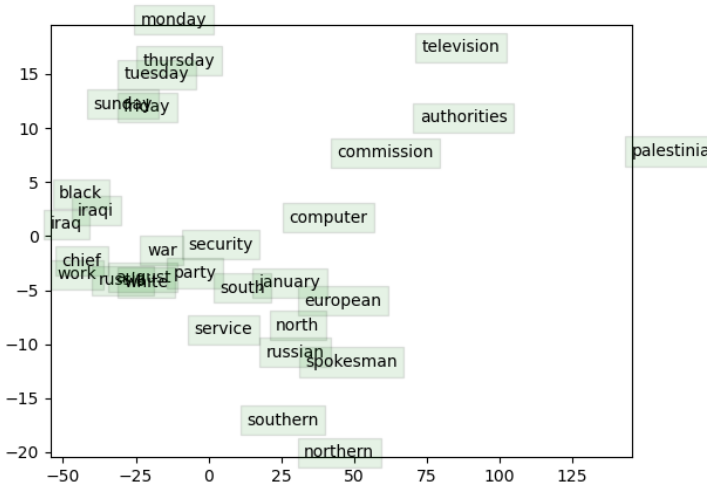


Figure 3: fasttext visualization

3 Vector Representation of Sentences (12+4)

Based on word representation which we learned from Questions 1 and 2, we will represent sentences by averaging vectors of words consisting of sentences (Bag of words). Skeleton code is provided on `sentence/representing_sentence_vectors.py` file. Every method and function is presented for you. What you are supposed to do is just run those codes and write down your answer.

(a) Tokenization splits a sentence (string) into tokens, rough equivalent to words and punctuation. For example, to process the sentence 'I love New York', the given sentence needs to be tokenized to ['I', 'love', 'New', 'York']. Many NLP libraries and packages support tokenization, because it is one of the most fundamental steps in the NLP pipeline. However, there is no standard solution that every NLP practitioner agrees upon. Let's compare how different NLP packages tokenize sentences.
What is the main difference between two methods, `word_tokenize` and `WordPunctTokenizer`? (state within one line) **(4pts)** **Answer:** `WordPunctTokenizer`의 경우 어포스트로피(`'`), 하이픈(`-`) 등 점(`punctuation`)을 독립적으로 토큰화하고 반대로 `word_tokenize`의 경우 구두점을 단어 또는 `Character`의 일부분으로 포함시켜 표현한다.

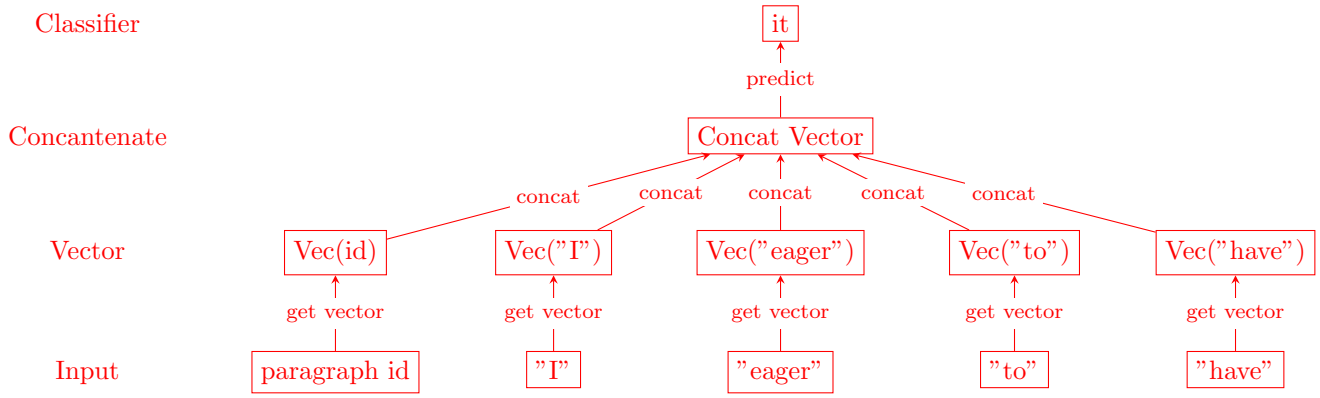
(b) Stop words are the words in a stop list which are filtered out (i.e. stopped) before or after processing of natural language data (text). Let's check out the English stopwords list of NLTK by running the code in the `representing_sentence_vectors.ipynb`.
After running the code and checking out the list, state **TWO** reasons why stopwords removal is useful in preprocessing. **(4pts)** **Answer:**
1) 더 의미 있는 단어가 자주 나올 수 있도록함(France-Paris not France-The). 의미있는 단어를 더 남겨 모델 향상을 꽤할 수 있음 2) 데이터 사이즈가 줄어들어 속도 향상을 꽤할 수 있음

(c) Find three sentences where s_1 and s_2 have similar meanings and s_1 and s_3 have opposite meanings, while cosine distance between (s_1, s_2) is longer than (s_1, s_3) As an example, s_1="I like everything of this movie. The only thing I do not like is the cast." is closer to s_3="I do not like everything of this movie. The only thing I like is the cast." than to s_2="I love all about this movie." in the vector space. Please find a different example that satisfies the above. Once you have found your example, please give a possible explanation for why this counter-intuitive result may have happened. **(4pts)**

Answer: s_1과 s_3의 경우 stop word를 제거하고 나면 동일한 Token만 남음. counter_intuitive_sentence function은 문장을 각 토큰으로 나누고, 각 토큰의 vector값을 vocabulary에서 꺼내 옴. 이후 단어 갯수만큼 나눠서 평균을 내기 때문에 s_1과 s_3는 동일한 embedding을 가지며, 더 큰 cosine similarity를 가짐.
또한 Token간 Sequence가 고려되지 않음

(d) **(Bonus)** What could we do beyond bag of words to resolve the issues presented at the problem (c)? Propose your own model that can represent sentences only using multi-layer perceptron (MLP) that can process variable-length sentences. Simply describe the idea with writing (and drawing) in a few

lines. **(4pts)**
(HINT: if you know CNN, use MLP in similar manner)
Answer: **1)** stopword를 제거하지않음 ∵ **2)** 각 문장을 Tokenize함. 또한 각 문장의 고유 id를 만들고 이에 해당하는 vector를 초기화함. 이때 vector는 단어의 vector 차원과 동일하게 함 **3)** (1D CNN) 문장의 각 Token을 sliding window이용, 윈도우 크기에 맞게 group으로 만듬. 이 벡터들을 concantenate함. 그 후 문장 id에 해당하는 vector를 concatenate함. **4)** 2)에서 생성된 토큰을 input으로 하여, 윈도우 사이즈 다음에 나오는 단어를 예측하는 방식으로 학습을 시킴
ex) "I eager to have it all in this room"



4 Document Retrieval (20+4)

Information Retrieval involves the process of obtaining information from a system based on an information need. One example is to obtain documents relevant to a user query from a large corpus of documents. The vector space model models these documents as vectors that represent these documents. A query can be represented as another vector. We can now find similar documents to a query by using some form of distance in the vector space, like cosine similarity:

$$\text{cosine similarity} = \cos(\theta) = \frac{\mathbf{q}.\mathbf{d}}{\|\mathbf{q}\|\|\mathbf{d}\|}$$

where $\mathbf{q}$ and $\mathbf{d}$ are the query and document vectors respectively.

How are these vector space models built? Let us represent a corpus of documents as a term-document matrix. A term-document matrix is a mathematical matrix that describes the frequency of terms that occur in a collection of documents. In a document-term matrix, rows correspond to documents in the collection and columns correspond to terms. The entries in the matrix can be defined in different ways (we will describe a few variations in this assignment question). A term document matrix allows for a way to index several documents against which a user query can be compared to fetch relevant documents.

Consider a corpus of documents:

```
I AM SAM
I LIKE SAM I AM
I LIKE GREEN EGGS AND HAM
```

4.1 Simple Word Counts

You can answer the following questions. Skeleton code is presented at `tfidf/document_retrieval.ipynb`.

(a) Create a term document matrix using simple word counts (number of times a word appears in a document) for all the documents. **(4pts)** **Answer:** 0은 생략됨

	I	AM	SAM	LIKE	GREEN	EGGS	AND	HAM
doc1	1	1	1					
doc2	2	1	1	1				
doc3	1			1	1	1	1	1

(b) Using the vector space model and cosine similarity, find the closest document to the user query "I LIKE EGGS" and their cosine similarity. **(4pts)** **Answer:** cosine similarity : 0.7071, 'I LIKE GREEN EGGS AND HAM'

(c) What are some issues you see of using counts to create the document vector? (one or two are enough.) **(4pts)**

Answer: 1) 겹치는 단어수로 하기 때문에 의미 있는 단어를 선정했다고 보기 어려움. 2) Document에 나오지 않은 단어가 Query에 나올때 처리가 어려움.

4.2 TF-IDF

One solution to this problem is to use the TF-IDF representation. TF-IDF involves multiplying the term frequency (which can be the raw counts) with an inverse document frequency (idf). The inverse document frequency is a measure of how much information the word provides, i.e., if it is common or rare across all documents. It is the logarithmically scaled inverse fraction of the documents that contain the word (obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient).

$$\text{idf}(t) = \log_2(\frac{\text{Number of documents in the corpus}}{\text{Number of documents } t \text{ appears in}})$$

(a) Now instead of using the raw counts, use TF-IDF for each entry in the term-document matrix. Using the vector space model and cosine similarity find the closest document to the user query "I LIKE EGGS" for the new index **(4pts)**

Answer: score : 0.2983, 'I LIKE GREEN EGGS AND HAM'

	I	AM	SAM	LIKE	GREEN	EGGS	AND	HAM
doc1	0	1	1	0	0	0	0	0
doc2	0	1	1	0.4150	0	0	0	0
doc3	0	0	0	0.4150	2	1	2	2
query	0	0	0	0.4150	0	1	0	0

(b) How does TF-IDF solve the issue quoted in 4.1? **(4pts)** **Answer:** 모든 문서에서 동일하게 발생하는 단어들은 중요하지 않은것으로 처리함으로서 좀더 의미 있는 단어를 기반으로 유사도를 측정함

(c) **(Bonus)** Instead of using cosine similarity we could also use the L2 distance. Implement the code using L2 distance. Based on the result, state which similarity function (L2 or cosine) would work better here. **(4pts)** **Answer:** 여기에선 cosine similarity를 사용하는게 좋을 것으로 예상함. 왜냐하면 L2의 경우 의미가 비슷한 문장이 더 큰 distance로 표현됨