

[과제 4] 문장 유사도 계산을 워드 임베딩 기법 등 다양한 방법으로 구현

- 입력: 한글 문장 1개
- 방법: 형태소 분석기 2가지와 WPM, 총 3가지 이상의 방법으로 실험

가장 정확도가 높은, 가장 실행 속도가 빠른 방법이 무엇인지, 각 방법들의 장점과 단점, 문제점 및 해결방안은 무엇인지를 비교/실험

- 제출물: 소스코드 및 실행파일, 보고서(실행속도 및 실행화면 스샷 포함)
- 3~5분 내외 동영상 제작(유튜브에 업로드) — 구현 방법 설명 및 실행 내용

<참고1>

입력문장과 말뭉치의 각 문장들을 워드 임베딩 기법 등을 이용하여 문장 벡터를 만들고, 벡터 유사도 계산 기법을 이용하는 방법을 추가한 경우 가산점을 부여함.

<참고2>

실행속도가 매우 빠른 새로운 방법을 사용하여 구현한 경우에 가산점을 부여함.

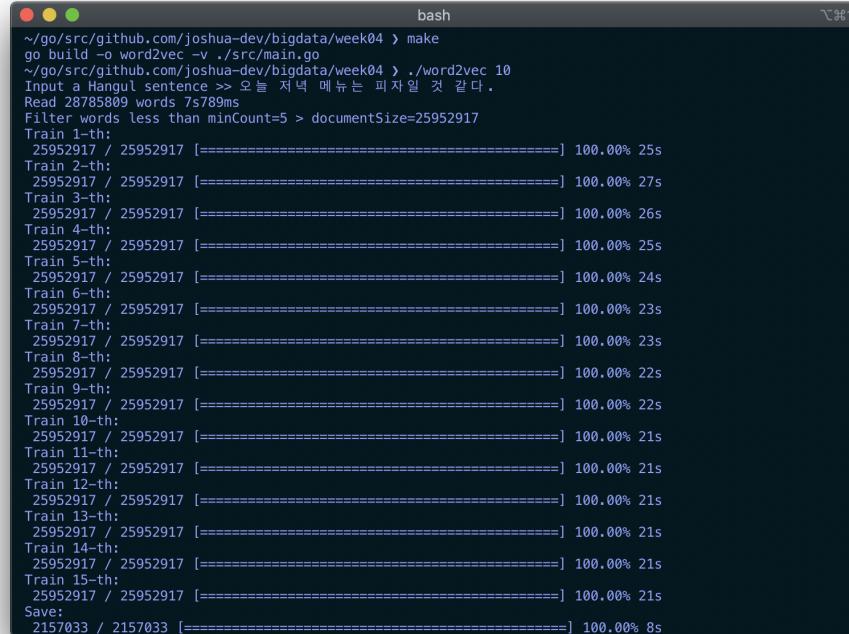
멀티 쓰레딩 기법, DBMS 등 다양한 기법 사용 가능.

- 제출하는 모든 실습 및 과제의 소스 코드는 아래 GitHub 경로에 있습니다.

<https://github.com/joshua-dev/bigdata>

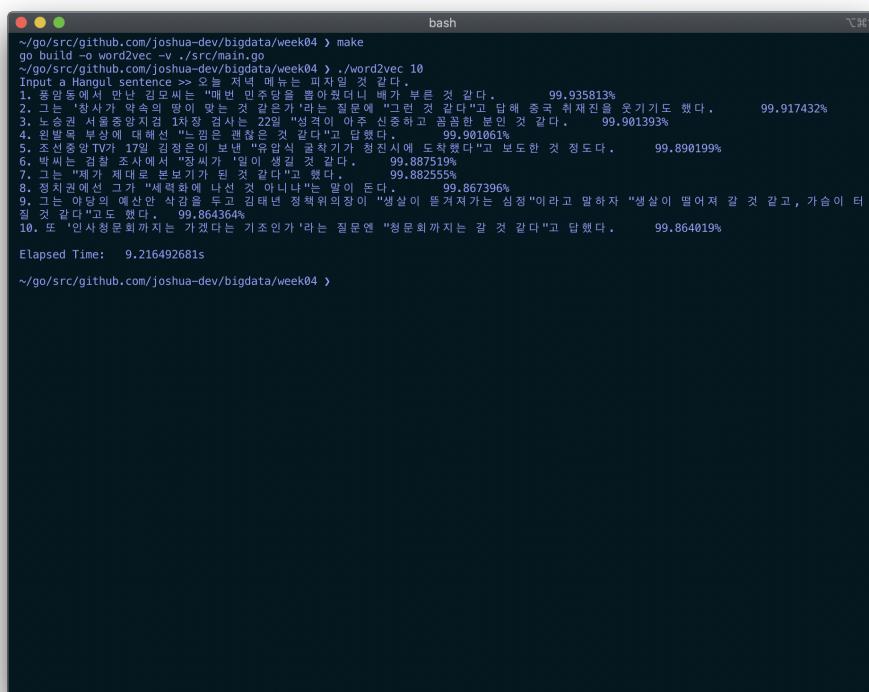
실행 결과

- 모델 훈련



```
~/go/src/github.com/joshua-dev/bigdata/week04 > make
go build -o word2vec -v ./src/main.go
~/go/src/github.com/joshua-dev/bigdata/week04 > ./word2vec 10
Input a Hangul sentence >> 오늘 저녁 메뉴는 피자일 것 같다.
Read 28785809 words 7s789ms
Filter words less than minCount=5 > documentSize=25952917
Train 1-th:
25952917 / 25952917 [=====] 100.00% 25s
Train 2-th:
25952917 / 25952917 [=====] 100.00% 27s
Train 3-th:
25952917 / 25952917 [=====] 100.00% 26s
Train 4-th:
25952917 / 25952917 [=====] 100.00% 25s
Train 5-th:
25952917 / 25952917 [=====] 100.00% 24s
Train 6-th:
25952917 / 25952917 [=====] 100.00% 23s
Train 7-th:
25952917 / 25952917 [=====] 100.00% 23s
Train 8-th:
25952917 / 25952917 [=====] 100.00% 22s
Train 9-th:
25952917 / 25952917 [=====] 100.00% 22s
Train 10-th:
25952917 / 25952917 [=====] 100.00% 21s
Train 11-th:
25952917 / 25952917 [=====] 100.00% 21s
Train 12-th:
25952917 / 25952917 [=====] 100.00% 21s
Train 13-th:
25952917 / 25952917 [=====] 100.00% 21s
Train 14-th:
25952917 / 25952917 [=====] 100.00% 21s
Train 15-th:
25952917 / 25952917 [=====] 100.00% 21s
Save:
2157033 / 2157033 [=====] 100.00% 8s
```

- 유사도 계산 및 실행 시간



```
~/go/src/github.com/joshua-dev/bigdata/week04 > make
go build -o word2vec -v ./src/main.go
~/go/src/github.com/joshua-dev/bigdata/week04 > ./word2vec 10
Input a Hangul sentence >> 오늘 저녁 메뉴는 피자일 것 같다.
1. 풍암동에서 만난 김모씨는 "매번 민주당을 봤던더니 배가 부른 것 같다. 99.935813%
2. 그는 청사가 악속으로 땅이 맞는 것 같은가 '라는 질문에 "그런 것 같다"고 답해 중국 취재진을 웃기기도 했다. 99.917432%
3. 노승권 서풀중앙지점 1차장 겸사는 22일 "성격이 아주 신중하고 꼼꼼한 분인 것 같다. 99.901393%
4. 임병우 부상에 대해선 "느낌은 괜찮은 것 같다"고 답했다. 99.901061%
5. 조선중앙TV가 17% 김정은이 보낸 유익식 글하기가 청진시에 도착했다"고 보도한 것 정도다. 99.890199%
6. 박씨는 김찰 조사에 대해 "정씨가 '몇 이 생길 것 같다. 99.887519%
7. 그는 청사에 차례로 블보드를 펼쳐놓고 된 것 같다"고 했다. 99.882555%
8. 경제권에서 고문 "세계화에 힘쓰는 것 아닙니다"는 말이 든다. 99.867396%
9. 그는 는 그의 예산안 석류를 두고 김태년 정책위의장이 "생살이 물거제가는 섬정"이라고 말하자 "생살이 떨어져 갈 것 같고, 가슴이 터질 것 같다"고도 했다. 99.864364%
10. 또 '인사 청문회까지는 가겠다는 기조인가 '라는 질문엔 "청문회까지는 갈 것 같다"고 답했다. 99.864019%
Elapsed Time: 9.216492681s
~/go/src/github.com/joshua-dev/bigdata/week04 >
```

Installation

```
$ make install
```

Run

```
// n means the number of high similarity sentences to be printed.  
// ex) ./word2vec 10  
$ make  
$ ./word2vec n
```

구현 방법 및 결과 분석

다음과 같은 순서로 워드 임베딩 기법을 사용한 문장 유사도 알고리즘을 만들었다.

모델의 학습 세대는 15회이고, 학습하는 데 걸리는 시간은 소요 시간에서 제외했다.

(각 세대 당 평균 학습 시간은 25초 내외이다.)

1. 사용자 입력 문장과 KCC 말뭉치로 워드 임베딩 구현체 중 word2vec 모델을 학습시키고 단어 벡터를 생성한다.
2. 시간 측정을 시작하고 생성된 단어 벡터를 로딩하여 자료구조에 저장한다.
3. KCC 말뭉치의 각 문장과 사용자 입력 문장을 ②의 단어 벡터를 사용하여 문장 벡터로 만들고 자료구조에 저장한다.
4. ③의 자료구조의 각 문장 벡터들과 사용자 입력 문장을 나타내는 문장 벡터를 고루틴 (goroutine) 을 이용한 멀티 쓰레딩 기법으로 빠르게 유사도를 계산하고 채널에 저장한다.
5. 최대 힙 (max heap) 을 생성하고 채널의 각 구조체에 접근하여 최대 힙에 push한다.
6. ⑤의 최대 힙에서 ⑥ 번 pop하여 유사도가 가장 높은 상위 n개 문장을 출력하고 시간 측정을 종료한다.

구현 상세

1. word2vec

word2vec은 Google의 Mikolov 등이 개발한 CBOW 방식과 Skip-gram 방식의 워드 임베딩 구현체이다.

이 라이브러리는 기본적인 임베딩 모형에 subsampling, negative sampling 등의 기법을 추가하여 학습 속도를 향상시켰다.

이번 과제에서는 word2vec의 go 구현체인 wego (Word Embedding in Go) 라이브러리를 사용했다.

[wego GitHub](#)

wego는 다음과 같이 각 단어들을 벡터의 형태로 변환한다.

```
<word> <value1> <value2> ...
```

word2vec의 장점은 각 단어들이 가지는 의미를 보존하기 때문에, 아래와 같은 연산이 가능하다는 것이다.

```
Vector("King") - Vector("Man") + Vector("Woman") = Vector("Queen")
```

- wego를 이용해 KCC 말뭉치와 사용자 입력 문장을 학습시키는 train.go

```
// Train trains word vectors using wego APIs.  
func Train(data, output string) error
```

2. 생성된 단어 벡터들을 로딩하여 map에 저장하는 main의 일부

```
vectors := make(map[string]vector.Vector)
outputFile, err := os.Open(outputData)
if err != nil {
    panic(err)
}

defer outputFile.Close()

scanner := bufio.NewScanner(outputFile)

for scanner.Scan() {
    txt := strings.Split(strings.TrimSpace(scanner.Text()), " ")
    word := txt[0]
    data := txt[1:]
    vector := vector.New(data)
    vectors[word] = vector
}
```

3. 2 의 단어 벡터들로부터 KCC 말뭉치와 입력 문장을 문장 벡터로 변환

단어 벡터로부터 문장 벡터를 생성하는 데에는 다양한 방법이 있다.

문장을 구성하는 단어의 단어 벡터를 단순히 더할 수도 있고, 각 단어들의 출현 빈도를 고려하여 가중치를 곱해서 더하는 SIF 방식도 있으며, RNN을 사용하여 가중치를 학습시키는 방법도 있다.

이 과제에서는 다음과 같이 각 문장을 구성하는 단어들의 단어 벡터를 더하는 방식으로 문장 벡터를 생성하였다.

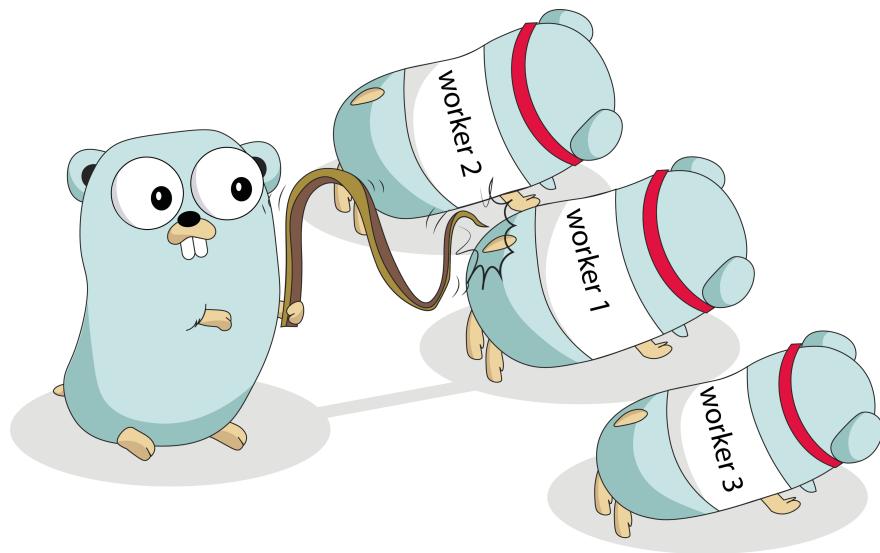
```
// Word2Sent returns the average vector of the given vectors.  
func Word2Sent(vectors []Vector) Vector
```

- KCC 말뭉치의 각 문장들을 문장 벡터로 변환하고 slice에 저장하는 main의 일부

```
var originVector vector.Vector  
  
scanner = bufio.NewScanner(inputFile)  
  
for scanner.Scan() {  
    sentence := strings.TrimSpace(scanner.Text())  
    words := strings.Split(sentence, " ")  
    wordVectors := make([]vector.Vector, len(words))  
  
    for index, word := range words {  
        wordVectors[index] = vectors[word]  
    }  
  
    sentenceVector := vector.Word2Sent(wordVectors)  
  
    if sentenceVector != nil {  
        if sentence == origin {  
            originVector = sentenceVector  
        } else {  
            temp := sentenceNvector{  
                sentence: sentence,  
                vector: sentenceVector,  
            }  
            sentenceNvectors = append(sentenceNvectors, temp)  
        }  
    }  
}
```

4. KCC 말뭉치의 문장 벡터들과 입력 문장의 문장 벡터를 비교

- goroutine



고루틴은 Go 런타임이 관리하는 논리적 (혹은 가상) 경량 쓰레드인데, OS 쓰레드보다 훨씬 가볍게 동시성 처리를 구현하기 위해 만들어졌다.

KCC 말뭉치로부터 생성된 각 문장 벡터들마다 고루틴을 하나씩 생성하고 동시에 입력 문장 벡터와 유사도를 계산하여 채널에 보냄으로써 동시성을 달성한다. (멀티 쓰레딩)

채널 (channel) 은 파이프와 같은 형태의 자료구조로 고루틴 간 통신에 사용된다.

두 벡터 간 유사도를 측정하는 데에도 다양한 방법이 있는데, 이 과제에서는 널리 쓰이는 방법인 코사인 유사도 (cosine similarity) 를 사용했으며 코사인 유사도를 아래와 같이 정의된다.

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

또한 멀티 쓰레딩과 더불어 논리적 CPU를 모두 계산에 참여시키기 위해 runtime 패키지의 GOMAXPROCS 함수를 호출했다.

- 두 문장 벡터 간 코사인 유사도를 계산하는 함수 Cos

```
// Cos computes the cosine similarity of the given vectors.
func Cos(v, u Vector) float64 {
    return v.dot(u) / (v.norm() * u.norm())
}
```

- goroutine을 이용해 병렬적으로 유사도를 계산하여 채널에 보내는 main의 일부

```
expressions := make(chan vector.Expr, len(sentenceNvectors))

var syncer sync.WaitGroup

syncer.Add(len(sentenceNvectors))

for index := 0; index < len(sentenceNvectors); index++ {
    go func(i int, ch chan vector.Expr) {
        defer syncer.Done()

        expression := vector.NewExpr(originVector, sentenceNvectors[i].vector, sentenceNvectors[i].sentence)
        ch <- expression
    }(index, expressions)
}

defer close(expressions)

syncer.Wait()
```

- runtime.GOMAXPROCS 함수를 호출하여 모든 논리적 CPU를 사용하게 하는 main의 일부

```
runtime.GOMAXPROCS(runtime.NumCPU())
```

5. 채널의 각 문장-유사도 구조체에 접근하여 최대 힙에 push

유사도가 가장 높은 순으로 정렬하는 방법도 있지만, 가장 높은 n개만 뽑아내는 데에는 최대 힙 (max heap) 가장 효율적이다.

최대 힙은 부모 노드가 자식 노드보다 항상 큰 값을 가지는 자료구조로, heap 인터페이스를 달성함으로써 구현할 수 있으며 heap 인터페이스는 Len, Less, Swap, Push, Pop 메소드를 정의함으로써 달성할 수 있다.

- 입력 문장과의 코사인 유사도를 기준으로 최대 힙을 구현한 maxheap.go

```
// MaxHeap is a max heap type.
type MaxHeap []vector.Expr

// Len implements the heap interface.
func (m MaxHeap) Len() int

// Less implements the heap interface.
func (m MaxHeap) Less(i, j int) bool

// Swap implements the heap interface.
func (m MaxHeap) Swap(i, j int)

// Push implements the heap interface.
func (m *MaxHeap) Push(element interface{})

// Pop implements the heap interface.
func (m *MaxHeap) Pop() interface{}
```

- 최대 힙을 생성하고 채널의 구조체를 heap push를 통해 최대 힙에 저장하는 main의 일부

```
maxHeap := maxheap.New()

heap.Init(maxHeap)

for i := 0; i < len(expressions); i++ {

    heap.Push(maxHeap, <-expressions)

}
```

6. 최대 힙에서 n번 pop하여 유사도가 높은 문장 출력 및 시간 측정

- 6 을 구현한 main의 일부

```
for i := 0; i < argc; i++ {
    expr := heap.Pop(maxHeap).(vector.Expr)
    fmt.Fprintf(os.Stdout, "%d. %s\n", i+1, expr)
}

elapsedTime := time.Since(startTime)

fmt.Fprintf(os.Stdout, "\nElapsed Time:\t%s\n\n", elapsedTime)
```

타 모델과의 비교 및 결과 분석

(학습 데이터는 모두 KCC 말뭉치로 동일)

- N-gram으로 유사도를 측정한 경우

```
~/go/src/github.com/joshua-dev/bigdata/week02/ngram master*
> make test
go test -v ./src/ngram/...
==== RUN ExampleBigram
---- PASS: ExampleBigram (0.00s)
==== RUN ExampleTrigram
---- PASS: ExampleTrigram (0.00s)
PASS
ok      github.com/joshua-dev/bigdata/week02/ngram/src/ngram    0.662s

~/go/src/github.com/joshua-dev/bigdata/week02/ngram master*
> make
go build -o ngram -v

~/go/src/github.com/joshua-dev/bigdata/week02/ngram master*
> ./ngram
2020/03/29 02:53:04
Invalid command-line arguments:
Usage: ./ngram [arguments]
The arguments are:
2: compare two Hangul strings using bigram.
3: compare two Hangul strings using trigram.
5: compare two Hangul strings using both bigram and trigram.

~/go/src/github.com/joshua-dev/bigdata/week02/ngram master*
> ./ngram 2
Start Hangul similarity test using Bigram.

첫 번째 문장 >> 슬픔은 슬픔대로 오게 하라.
두 번째 문장 >> 기쁨은 기쁨대로 가게 하라.
음절 Bigram에 의한 유사도 : 46.875000%

~/go/src/github.com/joshua-dev/bigdata/week02/ngram master* 14s
> ./ngram 3
Start Hangul similarity test using Trigram.

첫 번째 문장 >> 우리 눈에 다 보이진 않지만,
두 번째 문장 >> 우리 귀에 다 들리진 않지만
음절 Trigram에 의한 유사도 : 61.290323%

~/go/src/github.com/joshua-dev/bigdata/week02/ngram master* 21s
> ./ngram 5
Start Hangul similarity test using Bigram and Trigram.

첫 번째 문장 >> 아무런 기대 없이 맞이하고,
두 번째 문장 >> 아무런 기약 없이 헤어져도
음절 Bigram에 의한 유사도 : 53.125000%
음절 Trigram에 의한 유사도 : 48.387097%


~/go/src/github.com/joshua-dev/bigdata/week02/ngram master* 18s
>
```

- SPM 모델로 학습시켜 유사도를 측정한 경우 (50000개 문장으로 학습)

```

bash
}

trainer_interface.cc(267) LOG(INFO) Loading corpus: ./src/spm/KCC940_Korean_sentences_UTF8.txt
trainer_interface.cc(315) LOG(INFO) Loaded all 50000 sentences
trainer_interface.cc(330) LOG(INFO) Adding meta_piece: <unk>
trainer_interface.cc(330) LOG(INFO) Adding meta_piece: <s>
trainer_interface.cc(330) LOG(INFO) Adding meta_piece: </s>
trainer_interface.cc(330) LOG(INFO) Normalizing sentences...
trainer_interface.cc(384) LOG(INFO) all char count=825555
trainer_interface.cc(392) LOG(INFO) Done! 99.9502% characters are covered.
trainer_interface.cc(402) LOG(INFO) Alphabet size=124
trainer_interface.cc(403) LOG(INFO) Final character coverage=0.999502
trainer_interface.cc(435) LOG(INFO) Done! preprocessed 50000 sentences.
unigram_model_trainer.cc(129) LOG(INFO) Making suffix array...
unigram_model_trainer.cc(133) LOG(INFO) Extracting frequent sub strings...
unigram_model_trainer.cc(184) LOG(INFO) Initialized 121066 seed sentencieces
trainer_interface.cc(441) LOG(INFO) Tokenizing input sentences with whitespace: 50000
trainer_interface.cc(451) LOG(INFO) Done! 179546
unigram_model_trainer.cc(470) LOG(INFO) Using 179546 sentences for EM training
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=0 size=66987 obj=14.2945 num_tokens=408743 num_tokens/piece=6.10912
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=60041 num_tokens=410596 num_tokens/piece=6.83859
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=45025 obj=13.0625 num_tokens=426450 num_tokens/piece=9.4714
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=44994 obj=13.0224 num_tokens=426771 num_tokens/piece=9.48951
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=0 size=33742 obj=13.1862 num_tokens=447472 num_tokens/piece=13.2616
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=33740 obj=13.1487 num_tokens=447499 num_tokens/piece=13.2632
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=0 size=25308 obj=13.364 num_tokens=469615 num_tokens/piece=18.5582
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=25308 obj=13.317 num_tokens=469612 num_tokens/piece=18.5581
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=18978 obj=13.5784 num_tokens=493236 num_tokens/piece=25.98909
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=18978 obj=13.5254 num_tokens=493233 num_tokens/piece=25.9897
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=0 size=14233 obj=13.8348 num_tokens=517979 num_tokens/piece=36.3928
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=14233 obj=13.7754 num_tokens=518018 num_tokens/piece=36.3956
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=0 size=11008 obj=14.0966 num_tokens=541137 num_tokens/piece=49.1943
unigram_model_trainer.cc(486) LOG(INFO) EM sub_iter=1 size=11008 obj=14.0302 num_tokens=541137 num_tokens/piece=49.1943
trainer_interface.cc(507) LOG(INFO) Saving model: BPE.model
trainer_interface.cc(531) LOG(INFO) Saving vocabs: BPE.vocab
Enter a Hangul sentence: >> 어느 날 미리 어서 블이 자랐다
1. 4%다: 66.666667%
2. 1%다: 66.666667%
3. 7%다: 66.666667%
4. 중앙선거관리위원회는 6일 보도자료를 배포하고 지원선거 및 국회의원 재·보선 사전투표를 8일부터 9일까지 실시한다며 별도의 신고는 필요 없으므로 선거권이 있는 사람이라면 전국 어느 사전투표소에서 투표할 수 있다고 밝혔다: 50.000000%
5. LAC다 쳐스 투수 류현진 이 지난달 27일 센프란시스코 차이언츠와의 원정경기에서 투구하고 있다: 50.000000%
6. 0: 50.000000%
7. 정세균 국무총리와 원내대표는 12일 오전 국회의사당 창간 광장에서 열린 2018 평창 동계올림픽 마스코트 제작식에서 수호랑과 반다비를 배경으로 차원봉사자 및 서동초등학교 어린이들과 파이팅을 외치고 있다: 50.000000%
8. 1%p다: 50.000000%
9. 0: 50.000000%
10. 1%로 인트 다: 50.000000%
Elapsed time: 8.875745s
(venv) ~@KMU/2020-1/bigdata/bigdata/week03 >
```

- word2vec 모델로 학습시켜 유사도를 측정한 경우 (2878만개 문장으로 학습)

```

bash
~/go/src/github.com/joshua-dev/bigdata/week04 > make
go build -o word2vec -v ./src/main.go
~/go/src/github.com/joshua-dev/bigdata/week04 > ./word2vec 10
Input a Hangul sentence: >> 오늘 저녁 매뉴는 차릴 것 같다.
1. 풍물동에서 만난 김모씨가 "매번 민주당을 봄아쳤더니 봄아쳤던 것 같다."고 봄아쳤던 것 같다. 99.935813%
2. 노승관은 서울 중앙지검 14차장을 겸임하면서 29일 "성적은 아주 신중하고 꼼꼼히 분임 것 같다."고 담해 중국 취재진을 웃기기도 했다. 99.917432%
3. 노승관은 서울 중앙지검 14차장을 겸임하면서 29일 "성적은 아주 신중하고 꼼꼼히 분임 것 같다."고 담해 중국 취재진을 웃기기도 했다. 99.917432%
4. 일별별로 부상에 대해선 "느낌은 괜찮은 것 같다"고 답했다. 99.901061%
5. 조선증인 TV가 17일 김정은이 보낸 "우암식 글쓰기기 형진시에 도착했다"고 보도한 것 정도다. 99.890199%
6. 박씨는 "제가 제대로 본보기가 된 것 같다."고 했다. 99.887519%
7. 그는 "제가 제대로 본보기가 된 것 같다."고 했다. 99.882555%
8. 정치권에선 그가 "세력화에 나선 것 아니냐"는 말이 돈다. 99.867396%
9. 그는 아당의 예산안 책갈을 두고 김태년 정책위의장이 "생살이 끌어져가는 심정"이라고 말하자 "생살이 떨어져 갈 것 같고, 가슴이 터질 것 같다"고도 했다. 99.864364%
10. 또 '인사 청문회까지는'가 겠다는 기조인가 '라는 질문엔 "청문회까지는 갈 것 같다"고 답했다. 99.864019%
Elapsed Time: 9.216492681s
~/go/src/github.com/joshua-dev/bigdata/week04 >
```

결과 분석

모델 평가

Aa Name	≡ N-gram	≡ SPM	≡ word2vec
유사도 판별 능력	↓	-	↑
처리 시간	↑	↓	↑

N-gram 모델은 학습 데이터 없이 공통 음절/어절의 갯수로만 유사도를 계산하기 때문에 실행 시간은 적게 걸리지만

단순히 공통 음절/어절의 갯수만 세는 방식이므로 의미적으로 유사한 문장을 만들어내기에는 다소 무리가 있어보인다.

SPM 모델은 주어진 데이터로 모델을 학습시켜 유사도를 측정한다는 점에서는 워드 임베딩 방식과 유사하지만 unsupervised tokenizer이기 때문에 처리 속도가 느린 편이고,

공통인 단어 토큰을 세어 유사도를 측정하기 때문에 워드 임베딩 방식에 비해 의미적으로 유사한 문장을 골라내는 능력은 떨어진다.

word2vec 모델은 단어를 벡터로 변환하는 과정에서 단어가 가진 의미를 보존하기 때문에 타 모델보다 의미적으로 더 유사한 문장을 찾아내기에 더 적합하다고 볼 수 있다.

또한 Hierarchical Softmax와 Negative Sampling을 통해 최적화함으로써 처리 시간도 크게 줄어들었다.

멀티 쓰레딩을 통한 시간 단축을 고려하더라도 SPM 모델이 50000개의 문장으로 학습하여 유사도를 측정하는 시간과 word2vec이 2878만개 문장으로 학습하여 유사도를 측정하는 시간의 차이가

1초가 채 되지 않는다는 점에서 word2vec 모델이 성능적으로 더 우수하다고 볼 수 있다.