# Artificial Intelligence

## Assignment2 : Word Clustering

2016025532 컴퓨터전공 심수정

### 1. Code Compile Version

모든 코드는 python3.6 version 을 기준으로 작성되었습니다. 따라서 python3 version 이 설치되어 있지 않다면 이를 설치할 필요가 있습니다.

"**python3 [filename]**"을 입력한다면 해당 코드를 컴파일, 실행할 수 있습니다.

### 2. Code Review

```
import time
import collections
from math import log

start_time = time.clock()

classes = []

with open("WordEmbedding.txt") as words:
    objects = words.read().split()
    objects = [objects[i: i + 2] for i in range(0, len(objects), 2)]

words = []
word_distance = []
for i in objects:
    i[1] = list(map(float, i[1].split(",")))
    words.append(i[0])
    word_distance.append(i[1])
```

start time 을 확인하고, WordEmbedding.txt 를 열어 data 를 꺼내옵니다. 이 때 data 는 objects 라는 list 에 저장되는데, [[word, position], [word, position], ...]의 형태로 저장하게 됩니다. 또한 word, position 을 각각 words, word_distance list 에 저장하고, position 을 float 들의 list 로 변경합니다.

```
threshold = 0.4
cluster_similarity = "cosine"
print(cluster_similarity, "similarity with threshold", str(threshold))
```

threshold 를 저장하고, cluster_similarity 에 어떠한 similarity 를 사용할 것인지(e.g. cosine similarity, euclidean similarity) 저장합니다. 그리고 어떠한 similarity 로 어떤 threshold 와 함께 clustering 을 진행할지 출력합니다.

```
def get_similarity_matrix(classes):
    matrix = []
    # item_set = []
    for i in classes:
        tmp = []
        # item_set.append(i[0])
        for j in classes:
            if classes.index(i) < classes.index(j):
                continue
            tmp.append(get_similarity(i, j))
        matrix.append(tmp)
    return matrix
```

similarity matrix 를 계산하여 얻습니다. 각 classese list 안에 담겨 있는 cluster 간의 similarity 를 계산하여 이 행렬에 집어넣습니다. 단, 이 행렬의 경우 하삼각행렬의 모양을 가집니다.


similarity 를 계산합니다. 우선적으로 item set x 와 item set y 가 같은 경우 무조건 euclidean 의 경우 0, cosine 의 경우 1 을 return 하고, 더이상 계산하지 않습니다.

후에 item set x, item set y 의 모든 item 조합을 보며 euclidean, cosine similarity 를 각각 계산하여 similarity list 에 넣어줍니다.

최종적으로, similarity list 에 아무것도 없다면 euclidean 의 경우 0, cosine 의 경우 1 을 return 합니다.

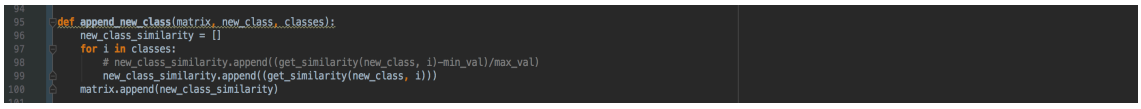

similarity 가 가장 높은 cluster 를 찾습니다. 그리고 classes list 에서 similarity 가 가장 높은 2 개의 cluster 를 return 하고, 이게 matrix 에서 어떤 row, 어떤 col 에 해당했는지를 return 합니다.


현재 word 가 몇번째 cluster 에 들어가있는지 찾아 index 를 return 합니다.


현재 matrix 를 출력합니다.


row, col 에 해당하는 matrix 의 row 를 삭제합니다.


row, col 에 해당하는 matrix 의 column 이 있다면, 삭제합니다.




새로운 cluster similarity 를 기존의 matrix 에 추가합니다.

```python
def list_to_set(list_content):
    set_content = set()
    for i in list_content:
        set_content.add(i[0])
    return set_content
```

cluster 내에 존재하는 word 만 set 으로 만들어 return 합니다.

```python
def get_entropy(result, topic_list):
    class_dict = {}
    for i in topic_list.keys():
        class_dict[i] = []

    for i in result:
        for topic, value in topic_list.items():
            for k in value:
                if i == k:
                    class_dict[topic].append(i)

    total = 0
    for i in class_dict.values():
        total += len(i)

    entropy = 0
    for i in class_dict.values():
        if len(i) == 0:
            continue
        entropy -= (len(i) / total) * log((len(i) / total), 2)

    return entropy
```

entropy 를 계산하는 함수입니다. Class 와 이에 해당하는 word 를 저장하는 dictionary 인 class_dict 를 만들어 이에 정확한 word, class 가 입력된 topic_list 의 key 만큼 key 를 빈 list 와 mapping 하여 넣어둡니다.
그리고 result 를 확인하여 각 class 에 해당하는 dictionary word list 에 들어가도록 해주고, 같은 cluster 내에 있는 word 가 몇개인지 확인합니다.
마지막으로 entropy 를 계산하여 return 합니다.

```python
def get_silhouette(result, others, words, word_distance):
    a = []
    b = []
    s = []
    for i in result:
        if len(result) > 1:
            a.append(sum([get_distance(i, j, words, word_distance) for j in result if j != i]) / (len(result) - 1))
        else:
            a.append(0)
        if len(others) > 0:
            b.append(
                min([sum([get_distance(i, j, words, word_distance) for j in k if j != i]) / len(k) for k in others]))
        else:
            b.append(0)
        if max(a[-1], b[-1]) != 0:
            s.append((b[-1] - a[-1]) / max(a[-1], b[-1]))
        else:
            s.append(0)
    return sum(s) / len(s)
```

silhouette 지표를 계산합니다. 우선 a(i)에 해당하는 값을 계산하여, list a 에 넣고, b(i)에 해당하는 값을 계산하여 b(i)에 넣습니다.
a(i), b(i)는 각각 같은 cluster 내의 다른 item 과 i 간 거리의 평균, 다른 cluster 의 속한 요소들 간 거리의 평균을 다른 모든 cluster 에 대해 구한 후, 가장 작은 값을 선택한 것입니다.
s(i)는 $(b(i) - a(i))/\max(a(i), b(i))$로 표현되는 값입니다. 이는 해당 cluster 내의 모든 word 에 대해 계산됩니다. 그리고 구해진 silhouette 값의 평균을 구해 return 합니다.

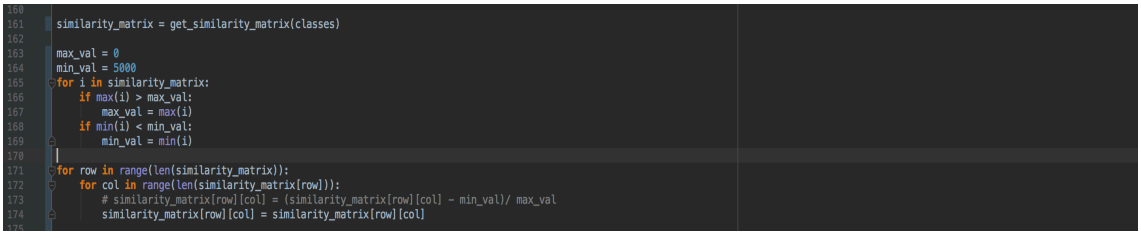
```python
def get_distance(item_x, item_y, words, word_distance):
    return sum(list(
        (word_distance[words.index(item_x)][index] - word_distance[words.index(item_y)][index]) ** 2 for index in
        range(len(word_distance[words.index(item_x)])))) ** 0.5
```

item_set_x, item_set_y 간의 euclidean distance 를 구하여 return 합니다.

```python
for i in objects:
    tmp = []
    tmp.append(i)
    classes.append(tmp)
```

cluster 를 [[item], [item], ...]의 형태로 표현하기 위해 objects 의 item 을 list

에 넣고, 이를 다시 cluster list 에 넣어줍니다.

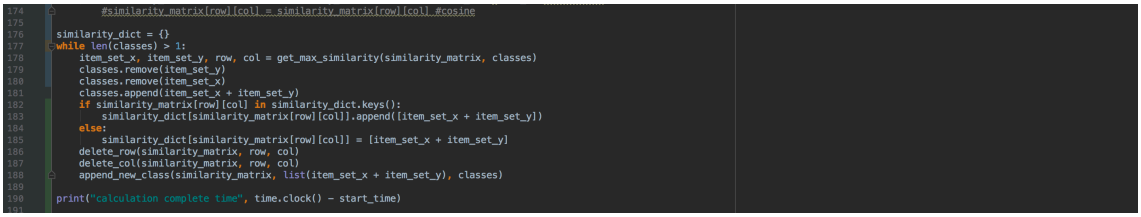


```
similarity_matrix = get_similarity_matrix(classes)

max_val = 0
min_val = 5000
for i in similarity_matrix:
    if max(i) > max_val:
        max_val = max(i)
    if min(i) < min_val:
        min_val = min(i)

for row in range(len(similarity_matrix)):
    for col in range(len(similarity_matrix[row])):
        # similarity_matrix[row][col] = (similarity_matrix[row][col] - min_val)/ max_val
        similarity_matrix[row][col] = similarity_matrix[row][col]
```

similarity matrix 를 먼저 구합니다. 그리고 euclidean 인 경우 max 값, min 값을 구하여 $similarity = {(similarity - min)}/{max}$의 형태로 만들어 0~1 사이의 값으로 나타날 수 있도록 합니다. cosine 의 경우는 그대로 유지합니다.



```
        #similarity_matrix[row][col] = similarity_matrix[row][col] #cosine

similarity_dict = {}
while len(classes) > 1:
    item_set_x, item_set_y, row, col = get_max_similarity(similarity_matrix, classes)
    classes.remove(item_set_y)
    classes.remove(item_set_x)
    classes.append(item_set_x + item_set_y)
    if similarity_matrix[row][col] in similarity_dict.keys():
        similarity_dict[similarity_matrix[row][col]].append([item_set_x + item_set_y])
    else:
        similarity_dict[similarity_matrix[row][col]] = [item_set_x + item_set_y]
    delete_row(similarity_matrix, row, col)
    delete_col(similarity_matrix, row, col)
    append_new_class(similarity_matrix, list(item_set_x + item_set_y), classes)

print("calculation complete time", time.clock() - start_time)
```

complete link clustering 을 수행합니다. 우선 가장 similarity 가 높은 cluster 2 개를 찾고, 이를 classes list 에서 제거합니다. 또한 이에 해당하는 similarity matrix 값도 지우고, classes, similarity matrix 에 이들이 합쳐진 cluster, 이에 대한 similarity 를 계산하여 넣습니다.

그리고 similarity dict 에 거리를 key 로 cluster list 안의 cluster 로 넣습니다. 이는 {distance : [[clsuter1], [cluster2]]}의 형태로 표현됩니다.

그리고 여기까지의 과정을 하는 동안 몇초의 시간이 소요되었는지 출력합니다.

```
similarity_dict = collections.OrderedDict(sorted(similarity_dict.items(), reverse=True))
#similarity_dict = collections.OrderedDict(sorted(similarity_dict.items()))

clustered_result = []
for i in similarity_dict.keys():
    if i <= threshold : #euclidean
    #if i >= threshold: #cosine
        for j in similarity_dict[i]:
            duplicated = False
            for k in clustered_result:
                new_set = list_to_set(j)
                if new_set.issubset(k):
                    duplicated = True
                    break
            if duplicated == False:
                clustered_result.append(list_to_set(j))

for i in words:
    exist = False
    for j in clustered_result:
        if i in j:
            exist = True
    if exist == False:
        clustered_result.append([i])
```

우선적으로 eulidean similarity 를 활용하는 경우라면 similarity 의 내림차순으로, cosine 을 이용하는 경우라면 similarity 오름차순으로 정렬합니다.

또한 similarity_dict 를 보며 similarity 가 threshold 를 만족하는 경우를 찾고, 이미 이 cluster 의 item 이 기존의 clustered_result 로 분류된 cluster 에 속하지는 않는지 확인 후 속하지 않는다면 clustered_result 에 추가합니다.

마지막으로, 기존의 word 들을 확인하며 현재 word 가 어떠한 cluster 에도

속해 clustered_result 에 들어가지 못했다면, word 만 존재하는 cluster 를 만들어 clustered_result 에 넣어줍니다.

```
total_word = 0
for i in clustered_result:
    total_word += len(i)
print("total word :", total_word)
print("total cluster :", len(clustered_result))
print("clustering complete time", time.clock() - start_time)
```

전체 단어가 다 들어갔는지 확인하기 위해 clustered_result 의 전체 단어수를 확인하고, cluster 의 수도 확인합니다. 그리고 이 과정을 위해 몇초의 시간이 소요되었는지 확인합니다.

```
with open("WordClustering.txt", "w") as output:
    for i in range(len(words)):
        output.write("{}\n{}\n{}\n".format(words[i], word_distance[i], get_cluster_number(words[i], clustered_result)))

with open("WordTopic.txt") as result:
    topic = result.read().splitlines()
    topic_list = {}
    for i in topic:
        if len(i) == 0:
            continue
        if i[0] == '(':
            topic_list[i[1:-1]] = []
            continue
        topic_list[list(topic_list.keys())[-1]].append(i)
```

처음에 읽은 word 와 vector 값을 clustering 을 통해 얻은 결과를 WordClustering.txt 를 열어 ₩n 단위로 씁니다.
WordTopic.txt 를 열어 정확한 cluster 당 word 들을 읽습니다. 그리고 이를 topic_list 에 {cluster : [word, word, ...]}의 형태로 집어넣습니다.

```
start_time = time.clock()
entropy_list = []
total = 0
for i in clustered_result:
    total += len(i)
    entropy_list.append(get_entropy(i, topic_list) * len(i))

for i in range(len(entropy_list)):
    entropy_list[i] = entropy_list[i] / total

new_entropy = sum(entropy_list)
print("entropy calculation time", time.clock()-start_time, " calculated entropy", new_entropy)
```

entropy 를 각 cluster 마다 계산해 해당 cluster 의 길이를 곱하고, 이의 평균을 전체적인 entropy 로 지정합니다.
$\sum^{entropy * length(cluster\ words)}/_{total\ words}$)가 전체적인 entropy 가 되는 셈입니다. 그리고 이를 entropy 를 계산하는데 걸리는 시간과 함께 출력합니다.

```
start_time = time.clock()
silhouette_list = []
total = 0
for i in clustered_result:
    total += len(i)
    silhouette_list.append(get_silhouette(i, clustered_result[0:clustered_result.index(i)] + clustered_result[clustered_result.index(i) + 1:-1], words, word_distance) * len(i))

for i in range(len(silhouette_list)):
    silhouette_list[i] = silhouette_list[i] / total

if len(silhouette_list) > 0:
    new_silhouette = sum(silhouette_list) / len(silhouette_list)
else:
    new_silhouette = -1

print("shihouette calculation time", time.clock()-start_time, " calculated shihouette", new_silhouette)
```

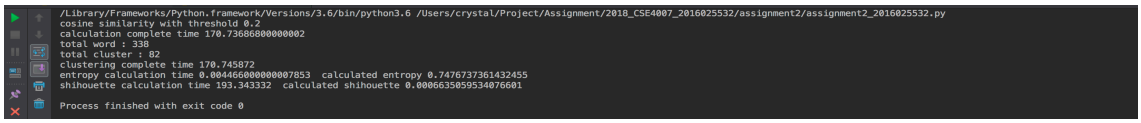
각 cluster 마다 silhouette 값을 계산합니다. 그리고 이를 silhouette list 에 넣고, $\sum^{silhouette * length(cluster\ words)}/_{total\ words}$)로 가중치 평균을 구합니다. 그리고 구해진 값을 출력하고, 이를 계산하는데 걸린 시간도 출력합니다.

## 3. Result

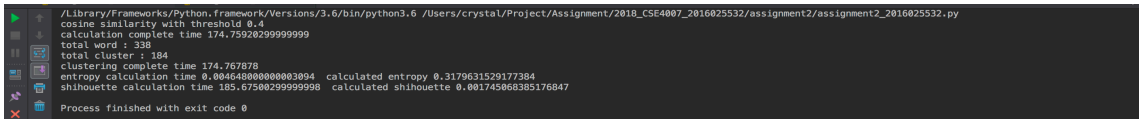
### 1) Cosine

cosine 의 경우 계산된 값이 클수록 similarity 가 크기에 threshold 가 클수록 더 유사한 cluster 만 허용합니다.

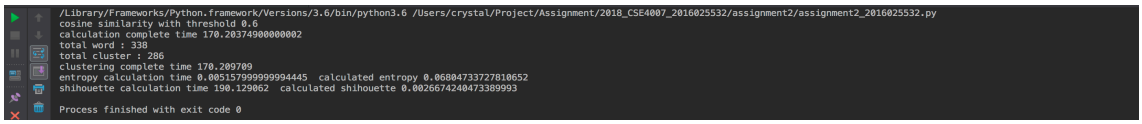
(1) threshold = 0.2



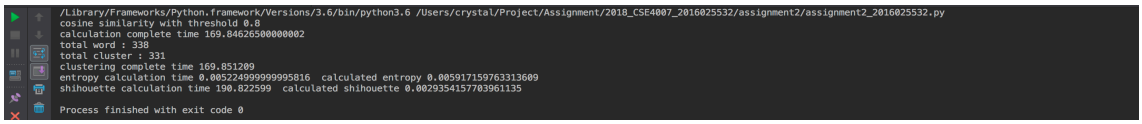

(2) threshold = 0.4




(3) threshold = 0.6



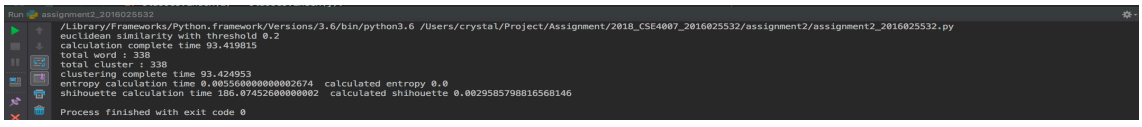

(4) threshold = 0.8




### 2) Euclidean

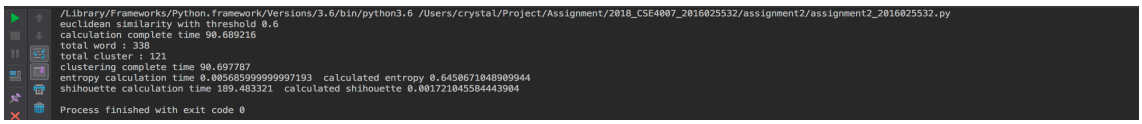
euclidean 의 경우 계산된 값이 similarity 가 클수록 작기에 threshold 가 작을수록 더 유사한 cluster 만 허용합니다.

(1) threshold = 0.2




(2) threshold = 0.4


(3) threshold = 0.6




(4) threshold = 0.8


### 3) Summary

| threshold | | Cosine Similarity | Euclidean Similarity |
|---|---|---|---|
| 0.2 | Cluster Number | 82 | 338 |
| | Entropy | 0.748 | 0.0 |
| | Silhouette | 0.000663 | 0.002959 |
| 0.4 | Cluster Number | 184 | 302 |
| | Entropy | 0.318 | 0.038 |
| | Silhouette | 0.001745 | 0.002782 |
| 0.6 | Cluster Number | 286 | 121 |
| | Entropy | 0.068 | 0.645 |
| | Silhouette | 0.002667 | 0.001721 |
| 0.8 | Cluster Number | 331 | 14 |
| | Entropy | 0.006 | 1.927 |
| | Silhouette | 0.002935 | 0.002549 |

### 4. Analysis

cosine 의 경우 threshold 가 높을수록 더 유사한 cluster 만 허용합니다. euclidean 의 경우 similarity 가 낮을수록 더 유사한 cluster 만 허용합니다. 그래서 cosine 에서는 threshold 가 높아질수록 entropy 가 감소하고, silhouette 가 증가하는 반면, euclidean 에서는 threshold 가 높아질수록 entropy 가 증가하고, silhouette 이 감소합니다.

보통 entropy 가 낮을수록 비슷한 성질의 것들끼리 clustering 이 잘 되었다고 말할 수 있습니다. 이 경우 cosine 에서는 threshold 가 0.8 경우, euclidean 에서는 threshold 가 0.2 인 경우가 가장 좋습니다. 이 지표만 봤을 때 가장 좋은 것은 entropy 가 0 인 Euclidean similarity, threshold 0.2 인 경우입니다. 이 경우도 entropy 와 마찬가지로 가장 clustering 이 잘 되었다고 보는 경우는 euclidean similarity, threshold 0.2 인 경우입니다.

silhouette 가 1 에 가까울수록 비슷한 성질의 것들끼리 clustering 이 잘 되

었다고 할 수 있습니다. 이 경우는 cosine 에서 threshold 가 0.8 인 경우, euclidean 에서 threshold 가 0.2 인 경우가 가장 좋습니다.

하지만 cluster 가 너무 세세하게 분할되어, word 1 개가 된다면 clustering 을 하는 의미가 없어집니다. 비슷한 Word 를 묶어 같은 특성의 아이를 찾고 싶은 건데, 그렇지 못하는 경우이기 때문입니다. 하지만 위에서 봤듯이, entropy, silhouette 지표에서는 모든 word 를 각각 하나의 cluster 로 묶는 Euclidean similarity, threshold 0.2 를 지지합니다.

그래서 cluster 의 수도 중요한 지표입니다. 이것만 고려하는 경우는 cosine 에서 threshold 0.2 인 경우가 가장 좋고, euclidean 에서 threshold 가 0.8 인 경우가 가장 좋습니다. 전체적으로 봤을 때는 압도적으로 Euclidean similarity, threshold 0.8 인 경우가 좋습니다.

이들을 종합하여 봤을 때, cosine similarity 에서 threshold 가 0.6 인 경우가 가장 좋다고 판단됩니다. Entropy 도 높지 않고, silhouette 지표도 높은 편이고, cluster 의 수가 높은 편인 것이 아쉽기는 하지만 그 이상이 되면 정확도가 떨어지리라 생각됩니다. 그래서 어느 정도의 정확도를 고려하며 clustering 을 하기 위해서는 이게 가장 좋다고 판단됩니다.