Programming Assignment 4

> Introduction to Information Retrieval and Text Mining
> R12725026 秦孝媛

1. **執行環境**：VS code
2. **程式語言**：python 3.10
3. **執行方式**：
    a. 打開 `pa4.py`
    b. 安裝需要的 packages，`pip install nltk`
    c. 執行 `python pa4.py`
    d. HAC 分群結果將儲存至 `./8.txt`、`./13.txt`、`./20.txt`
    - 有實作成功 **HEAP 加速分群**
4. **作業處理邏輯說明**：
    a. Package Import：引入需要的套件，下載 `stopwords` 列表

```
import os
import re
import numpy as np
import math
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
import numpy as np
# Download necessary NLTK packages
nltk.download('stopwords')
```

    b. Document Preprocessing：對於 document 進行 pa1 前處理，包含 tokenizing、過濾 stop words 以及 stemming 等等

```
# Function to preprocess documents
def process_document(text, stop_words, ps):
    text = re.sub(r'\d+', '', text)  # Remove digits
    translator = str.maketrans('', '', '''!"#$%&'()*+,-
```

```
    text = text.translate(translator)  # Remove punctua
    tokens = text.lower().split()  # Convert to lowerca
    tokens = [word for word in tokens if word not in st
    return [ps.stem(word) for word in tokens]  # Stemmi

# Load stop words
stop_words = set(stopwords.words('english'))

# Initialize document frequency dictionary and Porter S
document_frequency = {}
ps = PorterStemmer()

# Process documents and compute document frequency
data_dir = './data'
doc_vectors = []  # To store final normalized TF-IDF ve
for filename in os.listdir(data_dir):
    if filename.endswith('.txt'):
        filepath = os.path.join(data_dir, filename)
        with open(filepath, 'r', encoding='utf-8') as f
            tokens = process_document(f.read(), stop_wo
            unique_tokens = set(tokens)
            for token in unique_tokens:
                document_frequency[token] = document_fr

# Sort terms and build term index
sorted_terms = sorted(document_frequency.keys())
term_index = {term: idx for idx, term in enumerate(sort
```

c. Compute TF-IDF：引用 pa2 的處理，計算 normalized TF-IDF，將每一個文件檔案儲存成向量 `doc_vectors`

```
# Compute TF-IDF for each document
N = len(os.listdir(data_dir))  # Total number of docume
for filename in os.listdir(data_dir):
    if filename.endswith('.txt'):
        filepath = os.path.join(data_dir, filename)
        with open(filepath, 'r', encoding='utf-8') as f
            tokens = process_document(f.read(), stop_wo
```

```
            tf = {token: tokens.count(token) for token
            tfidf_vector = np.zeros(len(term_index))
            for term, freq in tf.items():
                df_t = document_frequency[term]
                idf_t = math.log10(N / df_t)
                tfidf_vector[term_index[term]] = freq *
            # Normalize
            norm = np.linalg.norm(tfidf_vector)
            if norm > 0:
                tfidf_vector /= norm
            doc_vectors.append(tfidf_vector)

# Convert doc_vectors to a numpy array
doc_vectors = np.array(doc_vectors)
```

d. Custom HEAP Data Structure：定義會使用到的 heap 相關函數

```
# Custom heap push operation
def heap_push(heap, item):
    heap.append(item)
    _sift_up(heap, len(heap) - 1)

# Custom heap pop operation
def heap_pop(heap):
    last_item = heap.pop()
    if heap:
        return_item = heap[0]
        heap[0] = last_item
        _sift_down(heap, 0)
    else:
        return_item = last_item
    return return_item

# Custom sift up operation
def _sift_up(heap, child_idx):
    while child_idx > 0:
        parent_idx = (child_idx - 1) >> 1
        if heap[child_idx][0] < heap[parent_idx][0]:
```

```
            heap[child_idx], heap[parent_idx] = heap[pa
            child_idx = parent_idx
        else:
            break

# Custom sift down operation
def _sift_down(heap, parent_idx):
    child_idx = 2 * parent_idx + 1
    while child_idx < len(heap):
        right_idx = child_idx + 1
        if right_idx < len(heap) and not heap[child_idx
            child_idx = right_idx
        if heap[parent_idx][0] > heap[child_idx][0]:
            heap[parent_idx], heap[child_idx] = heap[ch
            parent_idx = child_idx
            child_idx = 2 * parent_idx + 1
        else:
            break

# Custom heapify operation
def heapify(heap):
    n = len(heap)
    for i in reversed(range(n//2)):
        _sift_down(heap, i)
```

e. Perform HAC algorithm：

- 首先計算文件向量間的 cosine simialirity，因為向量已 normalized，所以直接透過 dot 來求得相似度
- 接著，我使用自定義的 heap structure 來管理 priority queue，從而找到相似度最高的文件 pairs
- 隨著 clustering 的進行，會不斷合併最相似的兩個 clusters，並更新 priority queue `P` 和 similarity matrix `C`
- 計算新相似度的 `calculate_new_similarity` 函數：linkage method 中我實作了 single-linkage、complete-linkage 和 average-linkage。
- 最後，當達到指定的 clustering number `K` 後，函數結束並 return 所有 merge 的結果 `A`，以及最終的群集 `clusters`

- 本次我使用 complete link 來計算

```
# Modify the cosine similarity function for normalized v
def cosine_similarity(vector1, vector2):
    # No need to divide by the norms, as the vectors ar
    return np.dot(vector1, vector2)

# Function to perform HAC with different linkage option
def hierarchical_agglomerative_clustering(doc_vectors, l
    N = len(doc_vectors)
    C = {n: {i: {'sim': cosine_similarity(doc_vectors[n
                 'index': i} for i in range(N) if i != n}
    I = np.ones(N, dtype=bool)
    P = {n: [] for n in range(N)}
    clusters = {n: [n] for n in range(N)}  # Initialize

    for n in P:
        P[n] = [(-value['sim'], value['index']) for val
        heapify(P[n])
    A = []

    while np.sum(I) > min(Ks):
        # Find the pair of clusters with maximum simila
        k1, k2 = None, None
        max_sim = -np.inf
        for n, pq in P.items():
            if I[n] and pq:
                # Extract the similarity and index sepa
                neg_sim, idx = heap_pop(pq)
                sim = -neg_sim  # Negate the similarity
                if sim > max_sim:
                    max_sim, k1, k2 = sim, n, idx

        if k1 is None:  # No more clusters to merge
            break

        A.append((k1, k2, max_sim))
```

```
        I[k2] = False  # Mark the merged cluster as ina
        P[k1] = []

        # Update the priority queues and similarity mat
        for i in range(N):
            if I[i] and i != k1:
                # Delete old similarities for k1 and k2
                P[i] = [(s, ind) for s, ind in P[i] if
                heapify(P[i])

                # Calculate new similarity for k1 based
                new_sim = calculate_new_similarity(C, i

                # Update similarities in C
                C[i][k1] = {'sim': new_sim, 'index': k1
                C[k1][i] = {'sim': new_sim, 'index': i}

                # Insert new similarity into the priori
                heap_push(P[i], (-new_sim, k1))

        # Reconstruct the priority queue for k1

        for i in range(N):
            if I[i] and i != k1:
                heap_push(P[k1], (-C[k1][i]['sim'], i))
                heapify(P[k1])

        # Update cluster membership
        clusters[k1].extend(clusters[k2])
        del clusters[k2]

    return A, clusters

# Helper function to calculate new similarity based on
def calculate_new_similarity(C, i, k1, k2, doc_vectors,
    if linkage == 'single':
        return min(C[i][k1]['sim'], C[i][k2]['sim'])
    elif linkage == 'complete':
```

```
            return max(C[i][k1]['sim'], C[i][k2]['sim'])
        elif linkage == 'average':
            return (C[i][k1]['sim'] + C[i][k2]['sim']) / 2
        else:
            raise ValueError("Unknown linkage type: {}".for

    # Run HAC and get clusters
    merge_history, _ = hierarchical_agglomerative_clusterin
```

f. Output results：根據指定的 K = 8, 13, 20，分別輸出分群結果

```
    # Function to extract clusters for a given K from merge
    def get_clusters_for_K(merge_history, K, total_document
        clusters = {i: [i] for i in range(total_documents)}
        for merge in merge_history[-(total_documents-K):]:
            k1, k2, _ = merge
            clusters[k1].extend(clusters[k2])
            del clusters[k2]
        return clusters

    # Function to save clusters
    def save_clusters(clusters, filename):
        # Write to file
        with open(filename, 'w', encoding='utf-8') as file:
            for cluster in clusters.values():
                # Sort the document indices within each clu
                sorted_cluster_indices = sorted(cluster)
                # Write each document index to the file, fo
                for doc_index in sorted_cluster_indices:
                    file.write(f"{doc_index + 1}\n")
                # Write an extra newline character after ea
                file.write("\n")

    clusters_k8 = get_clusters_for_K(merge_history, 8, N)
    clusters_k13 = get_clusters_for_K(merge_history, 13, N)
    clusters_k20 = get_clusters_for_K(merge_history, 20, N)

    # Save clusters for different Ks
```

```
save_clusters(clusters_k8, './8.txt')
save_clusters(clusters_k13, './13.txt')
save_clusters(clusters_k20, './20.txt')
```