# IRTM Homework4 Report

B10705054 劉知微

## 1 Execution Environment

The code is executed in Anaconda Jupyter Notebook.

## 2 Programming Language

The programming language used is Python 3.

## 3 Execution Method

To run the code in Anaconda, ensure that the required environment is set up and all necessary libraries are installed. The following Python modules are used in the code:

```
import re
import math
import numpy as np
import pandas as pd from nltk.stem
import PorterStemmer from collections
import defaultdict import os
from scipy.sparse import csr_matrix
```

Before execution, ensure the `data` folder, `stopwords.txt` files are prepared in the same directory as the code. The file paths should be defined as follows:

```
document_folder = "./data"
stopword_file = './stopwords.txt'
```

To execute the code in Jupyter Notebook, either click the "Run" button for individual cells or use the "Run All" option under the "Kernel" menu to execute all cells sequentially.

# 4 Workflow

## 4.1 Tokenization and TF-IDF Vectorization

This part follows the same process as Homework 2, which involves tokenization, computing document frequency (DF) for terms, and calculating inverse document frequency (IDF) values. The workflow also includes the computation of normalized TF-IDF vectors.

Since the code is identical to the implementation in Homework 2 (PA2), the details will not be further elaborated here. For reference, the resulting sparse TF-IDF vectors are stored as:

```python
term_df = defaultdict(int)
idf_dict = {}
sparse_tfidf_vectors = []
```

The main difference in this workflow is that the TF-IDF vectors are converted into a dense matrix for further processing. The code snippet below demonstrates this conversion:

```python

# Convert sparse vectors to a dense matrix
all_indices = sorted(idf_dict.values(), key=lambda x: x[0])
tfidf_matrix = np.zeros((len(documents), len(all_indices)))
for doc_idx, tfidf_vector in enumerate(sparse_tfidf_vectors):
    for term_idx, value in tfidf_vector.items():

        tfidf_matrix[doc_idx, term_idx - 1] = value

min_term_idx = float('inf')
for doc_idx, tfidf_vector in enumerate(sparse_tfidf_vectors):
    for term_idx, value in tfidf_vector.items():
        if term_idx < min_term_idx:
            min_term_idx = term_idx
```

## 4.2 Max Heap

To accelerate the running speed of Hierarchical Agglomerative Clustering (HAC), I chose to implement a max heap. This data structure efficiently manages similarity values and supports the operations required for clustering. The max heap includes multiple functions such as insertion, extraction, and heap property maintenance. The implementation is shown below:

```
class MaxHeap:
  def __init__(self):
     self.heap = []
  def _parent(self, index):

  def _left_child(self, index):

  def _right_child(self, index):

  def _heapify_up(self, index):

  def _heapify_down(self, index):

  def push(self, similarity, cluster1, cluster2):

  def pop(self):

  def peek(self):

  def size(self):

  def is_empty(self):
```

## 4.3 HAC Implementation

To implement Hierarchical Agglomerative Clustering (HAC), I first compute a similarity matrix for all documents using cosine similarity. The similarity matrix is a key component in determining the most similar clusters to merge at each step of the clustering process.

The code for computing the cosine similarity matrix is as follows:

```

def compute_cosine_similarity(matrix):
   N = matrix.shape[0]
   similarity_matrix = np.zeros((N, N))
   for i in range(N):
      for j in range(N):
         if i == j:
            similarity_matrix[i, j] = 1.0 # Cosine similarity with
               itself
```

```python
            else:
                dot_product = np.dot(matrix[i], matrix[j])
                norm_i = np.linalg.norm(matrix[i])
                norm_j = np.linalg.norm(matrix[j])
                if norm_i > 0 and norm_j > 0:
                    similarity_matrix[i, j] = dot_product / (norm_i * norm_j)
                else:
                    similarity_matrix[i, j] = 0.0
    return similarity_matrix
similarity_matrix = compute_cosine_similarity(tfidf_matrix)
```

The HAC (Hierarchical Agglomerative Clustering) algorithm is officially implemented as follows. Initially, each document is treated as its own cluster. A max-heap is then initialized, where the similarities between all pairs of documents are pushed into the heap. To avoid redundancy and self-similarity, only pairs where the index of one document is larger than the other are considered, as the similarity matrix is symmetric and diagonal elements represent self-similarity.

To group the documents into $K$ clusters, the termination criterion is set to $K$. The algorithm proceeds by repeatedly popping the highest similarity pair from the max-heap, which represents the two clusters with the strongest similarity. These two clusters are then merged into one. Since one of the clusters is now part of a merged cluster, it ceases to exist independently. The similarity between the newly merged cluster and the remaining clusters is recalculated.

The "complete-link" method is used to compute the similarity between the new cluster and others. This method considers the minimum similarity (indicating the longest distance) between all possible pairs of documents in the two clusters. Once calculated, the new similarity values are pushed into the max-heap, ensuring that the heap always contains updated similarities.

This process is repeated until the desired number of clusters $K$ is achieved.

The following Python code implements this approach:

```python
def hierarchical_clustering(tfidf_matrix, similarity_matrix, K, method='complete-link'):
    N = tfidf_matrix.shape[0]
    print(N)

    # Initialize clusters
    clusters = {i: [i] for i in range(N)}

    # Initialize custom heap
    max_heap = MaxHeap()
    for i in range(N):
        for j in range(i + 1, N):
            similarity = similarity_matrix[i, j]
            max_heap.push(similarity, i, j)

    while len(clusters) > K:
        # Extract the most similar clusters
        _, cluster1, cluster2 = max_heap.pop()

        # Ensure clusters are still valid
        if cluster1 in clusters and cluster2 in clusters:
            # Merge clusters
            clusters[cluster1].extend(clusters[cluster2])
            del clusters[cluster2]

            # Recalculate similarity with other clusters
            for other_cluster in list(clusters.keys()):
                if other_cluster != cluster1:
                    if method == 'single-link':
                        sim = max(
                            similarity_matrix[i, j]
                            for i in clusters[cluster1]
                            for j in clusters[other_cluster]
                        )
                    elif method == 'complete-link':
                        sim = min(
                            similarity_matrix[i, j]
                            for i in clusters[cluster1]
                            for j in clusters[other_cluster]
                        )
                    max_heap.push(sim, cluster1, other_cluster)

    return clusters
```

This implementation ensures efficient merging of clusters and recalculation of similarities, enabling the generation of hierarchical clusters for a given $K$.

## 4.4 Outputs

To save the clustering results, a function was implemented to output the results in the required format. The function ‘save_clusters‘ organizes the clusters and saves them into files, with each cluster separated by an empty line. Within each cluster, document IDs are sorted and written line by line for clarity. The clusters themselves are sorted by their identifiers for consistent formatting.

The process begins by performing hierarchical clustering for different values of $K$, specifically $K = 8$, $K = 13$, and $K = 20$. For each value of $K$, the clustering results are computed using the previously implemented ‘hierarchical_clustering‘ function. These results are then saved to corresponding files named ‘8.txt‘, ‘13.txt‘, and ‘20.txt‘.

The Python code below illustrates this process:

```python

def save_clusters(clusters, file_name):
    # Sort the clusters by the minimum document ID in each cluster
    sorted_clusters = {k: sorted(v) for k, v in clusters.items()}
    sorted_clusters = dict(sorted(sorted_clusters.items(),
        key=lambda x: min(x[1])))

    with open(file_name, 'w') as f:
        for cluster_id, doc_ids in sorted_clusters.items():
            for doc_id in doc_ids:
                f.write(f"{doc_id+1}\n")
            f.write("\n")
for K in [8, 13, 20]:
    clusters = hierarchical_clustering(tfidf_matrix,
        similarity_matrix, K)
    save_clusters(clusters, f"{K}.txt")
```

This approach ensures the results are well-organized and easy to interpret, meeting the requirements of the homework. Each file represents a specific clustering solution, making it straightforward to compare the output for different values of $K$.